

**School of Computing**

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES



**UNIVERSITY OF LEEDS**

---

# **An Analysis of Shadow Mapping Techniques for Rendering Coloured Shadows**

**Jaina Modisett**

**Submitted in accordance with the requirements for the degree of  
High-Performance Graphics and Games Engineering M.Sc.**

**2022/2023**


The candidate confirms that the following have been submitted:

Items	Format	Recipient(s) and Date
<i>MSc Project Report</i>	<i>Word document</i>	<i>Dr M Billeter, Dr H Carr (18/08/2023)</i>
<i>Application Source Code</i>	<i>URL</i>	<i>Dr M Billeter, Dr H Carr (18/08/2023)</i>
<i>Performance Times Spreadsheets</i>	<i>URL</i>	<i>Dr M Billeter, Dr H Carr (18/08/2023)</i>

Type of Project: **Exploratory Software**

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student) 

## Summary

In applications which require the rendering of shadowed geometry, shadow mapping remains the predominant technique used in real-time settings, especially in applications aimed at consumer hardware. These shadow maps are almost always represented as single-channel, depth maps which encode geometry visibility from the perspective of a light. This representation allows for a binary set of outcomes for each fragment rendered which is affected by a given light; each fragment either receives the effect of the light or does not. Being depth maps, usual image filtering techniques, such as bilinear filtering, cannot be used to soften or smooth the shadow boundaries. To soften the hard, aliased boundary lines present in these shadow maps, various techniques such as percentage closer filtering [1] have been developed.

Even soft shadows though are still utilising binary visibility information stored in a shadow map, and therefore exclude several physical properties of scene geometry. While the exclusion or simplification of some physical aspects of surfaces is to be expected in real-time settings, whether for high computational cost or low visual impact, there are some properties which can contribute greatly to the end result of renders while not being overly taxing.

This report explores one such property: the light transmission of surfaces. Specifically, this report will analyse three techniques for rendering coloured and partial shadows cast by translucent surfaces, along with two depth-only shadow mapping techniques. The first technique is Fillion, et al.'s translucent shadows, a technique notable for its use in StarCraft II [2], the second is coloured stochastic shadow maps [3], and the final technique, composited translucent shadows, is a novel approach building on top of translucent shadows.

## Table of Contents

Summary .....	iii
1. Introduction .....	1
1.1 Project Aim.....	1
1.1.1 Context.....	2
1.1.2 Terminology .....	2
1.1.3 Scope.....	3
1.1.4 Existing solutions in released products.....	3
1.2 Objectives.....	3
1.3 Deliverables.....	4
1.4 Ethical, legal, and social issues.....	4
2. Background Research.....	4
2.1 Literature Survey.....	4
2.1.1 Shadow rendering techniques .....	5
<i>Early shadow mapping</i> .....	5
<i>Ray traced shadows</i> .....	6
2.1.2 Transparency / translucency techniques .....	6
<i>Transparency</i> .....	7
<i>Order-independent transparency</i> .....	8
<i>Depth peeling</i> .....	9
<i>Stochastic transparency</i> .....	10
<i>Stochastic shadow maps</i> .....	11
2.1.3 Coloured shadow mapping techniques .....	12
<i>Translucent shadows</i> .....	13
<i>Coloured stochastic shadow maps</i> .....	15
2.2 Methods and Techniques.....	18
2.3 Choice of methods .....	19
3. Software Requirements and System Design.....	19
3.1 Software Requirements .....	20
3.2 System Design .....	22

4. Software Implementation .....	23
4.2 Shadow mapping technique implementations .....	23
4.2.1 Translucent shadows .....	23
<i>Modifications</i> .....	23
<i>Resource requirements</i> .....	25
4.2.2 Coloured stochastic shadow maps .....	26
<i>Modifications</i> .....	26
<i>Resource requirements</i> .....	26
4.2.3 Composited translucent shadows .....	27
<i>Leveraging sorted geometry for partitioning</i> .....	29
<i>Depth peeling as a partitioning method</i> .....	30
5. Software Testing and Evaluation .....	32
5.1 Evaluation Method .....	33
5.2 Testing Hardware .....	35
5.3 Results .....	35
5.3.1 Performance .....	35
5.3.2 Shadow Quality .....	40
5.3.3 Layered Transparency .....	42
6. Conclusions and Future Work .....	44
6.1 Conclusions .....	44
6.2 Future Work .....	45
References .....	45
Appendix A: Third-Party Software .....	47
Appendix B: Ethical Issues .....	48
Appendix C: Project GitHub Link and Raw Evaluation Data .....	48

# 1. Introduction

## 1.1 Project Aim

Real-time shadows have been a staple of computer graphics for decades and their popularity is not surprising. While lighting itself is important for conveying information about the shape, texture, and other properties of a surface, without shadows it is difficult to avoid an artificial, incomplete appearance, not to mention the important visual information that is communicated through shadows. Shadows allow objects' scale and relative positions to be understood at a glance even when viewing two-dimensional renders; however, the most commonly utilised shadow mapping techniques still have shortcomings. A notable point of weakness is the handling of transparent geometry. Handling transparency from the perspective of the camera is a non-trivial task itself, but modelling light transmission through transparent surfaces and storing this information in a format that is able to be used for rendering real-time shadows is an even more complex task. However, the ability to reflect this transmission opens many new possibilities, such as a stained-glass window casting its design over a scene.

The aim of this project is to investigate the performance and results of real-time techniques for rendering partial shadows cast by translucent surfaces, and more specifically, techniques of this type that support colour filtering by surfaces, resulting in coloured shadows. The two existing techniques under consideration are translucent shadows as described by Filion et al. in their 2008 presentation on rendering techniques used in StarCraft II [2] and coloured stochastic shadow maps (CSSMs) [3]. Both of these techniques have demonstrated themselves as capable of running in real time without a prohibitive amount of additional overhead but do each bring with them artifacts or inaccuracies that arise as a result of certain aspects of their approaches. Additionally, a new technique, composited translucent shadows, is considered which builds atop the previously mentioned translucent shadows by compositing multiple render passes of the scene. This new method is more expensive than both existing approaches under consideration, due to the larger and varying number of render passes it necessitates, but also allows for convincingly shaded opaque and transparent geometry (including multiple layers of transparent geometry) to be rendered without noise or other artifacts, avoiding the major pitfalls of both and providing results which are closer to the ground truth.

Performance in this report is considered as the average time to render each frame of a given scene (both total time and the time of specific steps required by the techniques) along with memory resource requirements of the approach. Artifacts or flaws that are present in the renders of some techniques will be highlighted, but since the severity of these aspects are to

some degree a matter of personal taste, this portion of the analysis will be limited to a discussion, weighing the overall results and performance of each technique against its respective weaknesses.

### 1.1.1 Context

I first considered coloured shadows when experimenting with the open-source game engine, Godot, and noticing that translucent and transparent objects both cast shadows as if they were fully opaque and did not receive shadows themselves when appropriate. In tackling this issue, I quickly realised that most any approach which encoded partial light transmission through translucent objects would be able to additionally encode preferential transmission of different colours of light by simply considering each colour channel separately.

Researching this topic led me to the two documented techniques mentioned prior, each of which achieved the desired effect to varying degrees. However, despite the demonstration of promising results, the adoption of these techniques seems limited.

Also, while both techniques each have their own benefits and drawbacks, it still seems there is a vacancy for a technique that produces less artifacts than CSSMs and is closer to the ground truth than translucent shadows, at the cost of more computation time.

### 1.1.2 Terminology

Some of the terms used in the report are more specific to computer graphics. The techniques discussed in this report are implemented within a rasterised, forward-rendering based environment, where *surfaces*, composed of *polygons* or *triangles* (also referred to as *geometry* in this report), which make up three-dimensional models, are resolved to specific pixels on the display surface (the window that is rendering the application) that they overlap with. A section of a polygon that overlaps with a particular pixel is referred to as a *fragment*, each of which is processed within the fragment shader to determine an output colour. This rendering is triggered by a *draw call* where a mesh (or meshes or subsections of meshes) are submitted for rendering. Associated draw calls are grouped into *render passes*.

Additionally, the terms *transparency* and *translucency* are used interchangeably in this report, as is common in computer graphics environments. In cases where authors have specifically chosen a term in their publications such as *translucent shadows* [2] or *coloured stochastic shadow maps* [3], the given name is used. Shadows cast by translucent surfaces are

also referred to as *translucent shadows* or *partial shadows*, and additionally *coloured shadows* in instances where the shadows are tinted by the surfaces they are being cast by.

### 1.1.3 Scope

Important to add is that this report is only considering techniques within the context of forward rendering. Many of the techniques discussed would also apply in deferred rendering environments, but in-depth consideration of the advantages and drawbacks of these settings when compared to forward rendering environments is considered out of scope for the purposes of this report.

Additionally, the effect of coloured shadows can be achieved through ray tracing or path tracing, but these rendering techniques themselves demand significantly more computation power when compared to traditional forward rendering techniques due to the ray-geometry intersection testing required by them, so are not considered in the more in-depth analysis of this report, though they are briefly discussed.

### 1.1.4 Existing solutions in released products

One of the techniques under consideration, translucent shadows [2], saw use in Activision Blizzard's *StarCraft II* (the cited source is a write up on various rendering techniques used in the game), but little widespread adoption apart from this release was found.

While the instances of coloured shadows usage seem limited, I did find coloured shadows listed among the features of major game engines such as Unity and Unreal.

Unreal offers a page in its documentation covering the process of implementing "Colored translucent shadows" in the engine [4], but offers this feature only using baked lighting (this feature is also unsupported when using Unreal Engine 5's new Lumen Global Illumination) or path tracing, making it unsuitable for consideration in this report.

Unity's implementation is similar, offering coloured shadows when utilising ray tracing and baked lighting (using the `_TransparencyLM` shader property), but making the feature unavailable otherwise [5].

## 1.2 Objectives

- Research existing real-time solutions to rendering partial shadows cast by transparent objects, with a particular focus on techniques which allow the casting of coloured



shadows, as well as the associated topics of the rendering of translucent surfaces themselves, and related shadow rendering techniques.

- Implement existing techniques which produce the translucent shadow effect being analysed.
- Develop a technique that addresses the shortcomings of the existing techniques.
- Design scenes which will allow the techniques to be compared against one another in a meaningful way and benchmark the performance of each technique when rendering those scenes.
- Analyse the results of the benchmarking along with the renders that result from each of the techniques.

### **1.3 Deliverables**

- A GitHub repository containing the source code of the application associated with this project.
- This MSc Project report.
- The raw performance data collected for each shadow mapping technique.

### **1.4 Ethical, legal, and social issues**

Due to the focus of this project being on rendering techniques, there are no ethical or social considerations that needed to be undertaken during the course of the project.

Additionally, no external company or organisation has stake in this project, so there are no concerns on this front either.

Additional details concerning specifics of third-party code, libraries, and tools, as well as a discussion of code management can be found in Appendix B.

## **2. Background Research**

### **2.1 Literature Survey**

During the review of relevant existing literature, most sources fell into one of two categories: writing on shadow rendering techniques and writing on the rendering of transparent surfaces. There is of course overlap between the two, and considering this topic's focus is on the intersection of these two areas, sources that fell into this overlap were of particular interest.

For the purposes of presenting this literature survey topics have been divided into these two categories and are discussed in an order so that more foundational research comes before research built atop it.

### 2.1.1 Shadow rendering techniques

Realtime shadow rendering techniques have been around for decades with the two most common techniques: shadow volumes [6] and shadow mapping [7], both being first written about in the 70s. Of these two techniques shadow mapping is by far the more popular, being the technique used in a great majority of real-time settings, particularly those targeting consumer hardware such as video games due to its cheap cost and efficient scaling with more complex scenes. Indeed, shadow mapping is the foundation for all of the techniques which are analysed more deeply in this report. Some techniques, such as shadow volumes, while significant, are not as applicable for the purposes of rendering translucent shadows, so aren't discussed in much depth.

#### *Early shadow mapping*

Lance Williams published "Casting Curved Shadows on Curved Surfaces" [7] in 1978, introducing what is now called shadow mapping. The premise introduced in that paper is the same one used in shadow mapping techniques today.

Shadow mapping takes advantage of the Z-buffer used to calculate surface visibility. The Z-buffer, along with a colour buffer is one of the most common framebuffer attachments in graphics. While framebuffers can have varying type and counts of attachments, a "standard" framebuffer consists of a colour buffer attachment and a Z-buffer (also called a depth buffer) attachment. While the colour buffer is straight forward, storing the colour contributions of fragments which are not discarded, the depth buffer instead stores their depth, or distance from the camera. In the standard depth-testing algorithm [8] fragments are discarded if their depth is greater than the depth of the current depth buffer value (i.e., they are farther from the camera than an already rendered surface), while if their depth is less than the current value, the fragments' colour and depth overwrite the existing values. This technique allows visibility from the camera's perspective to be calculated efficiently.

A light view matrix must first be calculated that converts points in world-space into the space of the light. Transforming geometry into this light-space and rendering it to a framebuffer with only a depth attachment records the depth of the closest surface to the light at the position associated with each texel [7]. This depth buffer is the titular shadow map.

After this initial depth-only pass the scene can be rendered as usual with one exception, for each fragment in the scene, its position is transformed into light-space, and its light-space depth is compared to the value stored in the shadow map. If a fragment is farther from the light than the value stored in the shadow map, it is in that surface's shadow and therefore does not receive lighting contribution from the associated light [7].

This algorithm is incredibly efficient. It scales linearly with the complexity of the geometry in the scene, and only ever requires one, depth-only render pass per shadow-casting light source.

### *Ray traced shadows*

More recently with the rise of real time ray tracing along with hardware optimised specifically for it, ray traced shadows have become more common, indeed they have been adopted in games and game engines (such as those mentioned previously) running on (high-end) consumer hardware in recent years and are capable of real-time frame rates. In particular, hybrid approaches utilising raytracing along with more traditional rasterization techniques have proven to be quite performant, as the rasterization can be used to determine primary-ray visibility, while shadow coverage on geometry can be resolved through more traditional ray-geometry intersection testing [9].

However, as previously stated, this report focuses on approaches utilising shadow mapping, so ray-traced shadowing techniques are not delved into much deeper. A deeper comparison of performance when rendering coloured, transparent shadows would be valuable, though out of scope for this project.

### **2.1.2 Transparency / translucency techniques**

As shadow mapping at its core is simply twice rendering a scene with an initial pass being rendered from the perspective of the shadow-casting light source, the techniques which are commonly used to render transparent geometry from the perspective of the camera, are also much the same techniques used when rendering more advanced shadow maps to add support for the partial shadows which are the focus of this report.

The techniques discussed in this section are far from the only means of rendering transparent geometry but are some of the most common and also the ones that are foundational to the shadow mapping techniques under consideration. With this in mind, it is

valuable to analyse some of these techniques and concepts in order to build up to the approaches which are being investigated in this report.

### *Transparency*

The depth-testing algorithm mentioned previously, is well suited for modelling visibility of opaque geometry, but when rendering translucent surfaces, this depth testing cannot be utilised in the same manner. If a transparent surface records its depth to the Z-buffer, it will hide geometry behind it that should be partially visible through it. Therefore, a different approach must be taken when rendering transparent surfaces. A common approach is to render opaque geometry in a separate render pass (with usual depth writes and comparisons) before rendering the transparent geometry in its own render pass (still depth testing against the opaque geometry, but not recording its own depth). Since depth testing alone cannot provide accurate results, a method of blending transparent surfaces' contributions with the colour information of geometry behind them is needed. Most commonly, the final colour of a given transparent fragment is determined by the following equation:

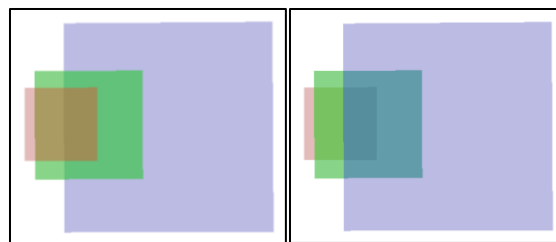
$$result_{colour} = source_{colour} * source_{weight} + destination_{colour} * destination_{weight} [8].$$

The source and destination colour are straightforward, representing the colour of a fragment that has just been rendered and the colour already stored in its output pixel respectively. The weights can vary depending on the implementation, but are most commonly calculated as:

$$\begin{aligned} source_{weight} &= source_{colour_{alpha}} \\ destination_{weight} &= 1 - source_{colour_{alpha}} [10]. \end{aligned}$$

Important to note in the original colour blending equation is that in successive draw calls which render additional translucent geometry to pixels which have already had a translucent fragment written to them, the  $result_{colour}$  becomes the  $destination_{colour}$  (and thus any information about already rendered translucent surfaces and their respective alpha values is lost). With this in mind, the resulting colour of a pixel which is influenced by multiple, overlapping translucent surfaces can be viewed as a composition of the above blending equation with itself. Additionally, because the sole factor determining the blending at each step of this equation is the alpha channel of the fragment currently being rendered, the final result is dependent on the render order of the translucent surfaces themselves. An example of the effects of incorrect ordering can be seen in Figure 2.1.2.A.

This dependency is a bit of a headache, but one which needs to be dealt with in some manner when dealing with translucent geometry (including when rendering the shadow maps this report is centred on). One approach is to simply order the translucent geometry by its proximity to the view position, but this can be an expensive operation, especially in complex scenes, and would need to be recomputed every time view position or scene geometry moves. Since the shadow mapping techniques discussed in this report involve rendering the geometry not only from the camera's perspective, but also from the light's perspective, for some of them, this proximity ordering must be done independently for each position, making it particularly unattractive.



**Figure 2.1.2.A.** The above images demonstrate incorrect (left) and correct (right) ordering of transparent surfaces. While it may not be immediately obvious that transparent surfaces are incorrectly layered when viewed in isolation, when viewed next to the correct result it becomes obvious.

The techniques under consideration approach this issue in varying ways. Filion, et al.'s translucent shadows, as well as composited translucent shadows that build upon them, simply perform this ordering to deal with the issue of overlapping translucent surfaces [2], but CSSMs use an alternative approach to rendering transparency, stochastic transparency [11], that doesn't require the surfaces to be ordered. This technique is part of a class of transparency rendering techniques referred to as order-independent transparency.

#### *Order-independent transparency*

There are numerous approaches to order-independent transparency (OIT), and they can be classed into exact and approximate approaches. Approaches of both types are explored in this report with stochastic transparency being an approximate OIT approach and depth peeling being an exact approach. As the name suggest, exact approaches are generally more expensive whether in memory demands or computational complexity (and usually both) but will result in an accurate final colour for a pixel resulting from multiple, overlapping translucent surfaces; approximate approaches on the other hand, are usually much faster than exact approaches, but

only estimate the final colour. Some approximate approaches can be scaled up and approach the ground truth (such as increasing the number of samples per pixel in stochastic transparency [11]), but this is not the case for all approximate approaches.

### *Depth peeling*

Depth peeling is a technique introduced in 2001 by Cass Everitt. It guarantees that for an arbitrary number of layers within a scene, they will be rendered accurately in an equal number of render passes. The basis of this technique revolves around the use of dual depth buffers to isolate individual layers into their own render passes, then compositing the layers together [12].

The initial render pass of this technique is very similar to rendering a scene full of opaque geometry, all of the geometry in the scene is rendered with depth writing and depth testing enabled (with less than being used as the depth function). This renders the layer of geometry closest to the view position and produces a matching depth buffer. The second pass uses new depth and colour output textures from the first pass and does the same rendering as the first pass with one notable exception, any geometry which fails a greater than depth test with the first pass's depth map is discarded. This effectively isolates the second layer of the scene and allows it to be rendered to its own colour texture. This process can be repeated for as many layers as is needed for a given scene (or until some stopping condition is met), with each render pass producing a new colour texture. The two depth buffers are reused, and which is being read from and written to is swapped each render pass.

After all layers have been rendered, they must be composited to produce the final image. The method suggested by Everitt is simply rendering each layer back-to-front to produce the final result (using the standard method described in the section on transparency) [12].

While an exact stopping condition is not described by Everitt in the original paper on depth peeling [12], the widespread adoption of depth peeling has led others to create various methods. Some implementations continue rendering until an empty layer is encountered (meaning all geometry has been peeled and rendered) [13]. Alternatively, one can simply set a fixed layer count, which additionally adds the benefit of stable performance since the number of render passes performed will be unchanging regardless of what geometry falls within the camera's view. This is acceptable in many circumstances, as subsequent layers have diminishing returns; each layer rendered reduces the influence of the layers behind it by an amount depending on the alpha values of its colour buffer [12]. This allows the algorithm to be stopped early without too much missing information in the final render, providing a sufficient number of

layers have been rendered. As with many techniques, the best approach varies depending on the requirements of a specific implementation.

Since the initial writing on depth peeling by Everitt, there have been significant advancements to the technique. One advancement of particular note is dual depth peeling [14]. Published by Bavoil and Myers in 2008, dual depth peeling nearly doubles the speed of the technique when compared to Everitt's original approach by simultaneously extracting both the front- and back-most layer with each render pass, and stopping once these layers are equal or overlapping. This allows dual depth peeling to half the cost of traditional depth peeling [14]. Another optimisation frequently used with depth peeling is to render all opaque geometry first and perform depth peeling only on transparent geometry which passes an additional depth test with the opaque geometry [13].

### *Stochastic transparency*

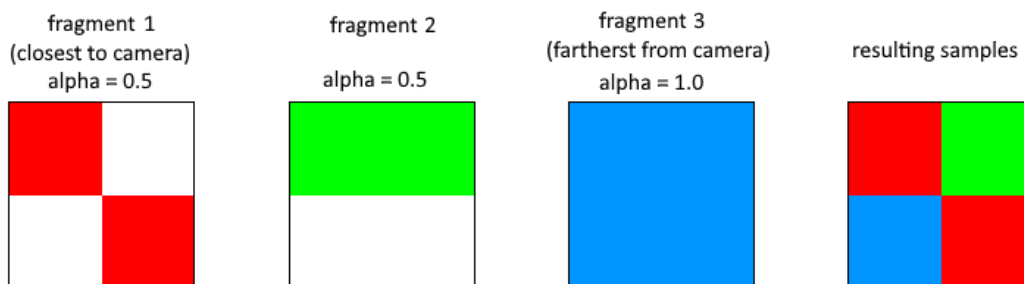
Some order-independent transparency techniques avoid rendering transparency all together by simply rendering transparent surfaces as opaque but selectively discarding fragments to simulate transparency.

The earliest approach of this type used pixel masks to selectively discard some fragments [15]; this approach is generally referred to as screen-door transparency. While these patterns are generally visible in a render, they do visually communicate translucency while still only rendering opaque fragments and therefore bypassing the need to sort or blend transparent geometry. There have been many extensions of this technique such as randomising the pixel masks on a per-polygon basis or multisampling approaches using subpixel masks (alpha to coverage) [11], but one of the more recent and more successful approaches is stochastic transparency. Introduced in 2010 by Eric Enderton, et al., stochastic transparency combines the two previously mentioned approaches, utilising randomised subpixel dithering patterns [11].

Stochastic transparency is accomplished by treating the alpha value of a fragment not as its contribution weight, but as it's chance to be rendered at all. This is to say, the fragments generated from a surface with an alpha value of 0.5 would only be rendered half of the time and discarded otherwise. If this is done with a single sample per fragment, the results will most likely be quite noisy and unappealing, but if multisampling is leveraged much better results can be achieved.

Given a finite number of samples per pixel in a buffer, the approach discussed by Enderton, et al. is to render each fragment to a randomly decided subset of samples within the pixel it occupies, with the size of the subset being weighted by its alpha value. Each individual

sample still performs the depth test as usual. For example, a fragment with an alpha value of 0.25 should on average be rendered to one fourth of the samples for the pixel that it covers, but if other transparent fragments are between it and the camera, it may lose its contribution to some or all of the samples it is rendered to due to the preceding fragments already occupying them [11]. In Figure 2.1.2.B, an example of a pixel with four samples, occupied by three fragments is given. It is worth noting that for fragments 1 and 2 in the figure they could potentially cover zero, one, two, three, or four of the samples in the pixel, but should on average be covering two samples. In this way and in the beneficial layout of which samples each fragment is rendered to, the figure is an ideal case.



**Figure 2.1.2.B.** An example of using stochastic transparency to resolve the colour of a multisampled pixel occupied by three fragments.

In "Stochastic Transparency," [11] Enderton, et al. address many improvements to the technique both in the random sampling method and other fronts, all to decrease noise while minimising the bias of the approach, but the above description summarises the core of the approach.

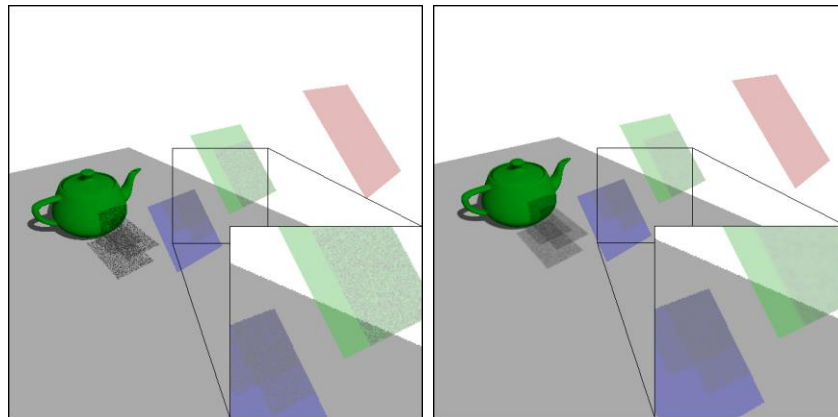
### *Stochastic shadow maps*

In the same paper that introduced stochastic transparency, Enderton, et al. introduced stochastic shadow maps. By utilising the same technique used to render transparent geometry in regular stochastic transparency (though rendering from the light's perspective and only to a depth buffer), you can render a shadow map which encodes information from multiple layers, weighted by their transparency [11].

This means the resulting shadow map can encode multiple layers and intensities of shadows in one depth buffer, an incredible capability. Utilising this shadow map to attempt to render hard shadows, using one shadow map sample per fragment to be shaded, will not give



very pleasing results due to the noise present in the shadow map from stochastic sampling; however, leveraging percentage-closer filtering (PCF) to soften the shadows, can give more pleasing results [11]. Given PCF is already very commonly employed for shadow maps, this means stochastic shadow maps could be utilised with very little overhead over a traditional shadow map. This said, stochastic shadow maps do require larger PCF kernels to overcome the noise, which will be slightly costlier and lead to shadows with blurrier boundaries; renders using stochastic shadow mapping with and without PCF can be seen in Figure 2.1.2.C.



**Figure 2.1.2.C.** On the left is an example of a stochastic shadow map being used to shade a scene without PCF to demonstrate the underlying method behind the approach (i.e., you can see the randomised dithering pattern on the surfaces), also of note is the proper (uncoloured) shading of both opaque and transparent surfaces. On the right is the same scene with PCF enabled.

Like with stochastic transparency, there is a discussion of improvements by Enderton, et al. to decrease the noise present in the shadows produced by the technique, and many of them are quite effective.

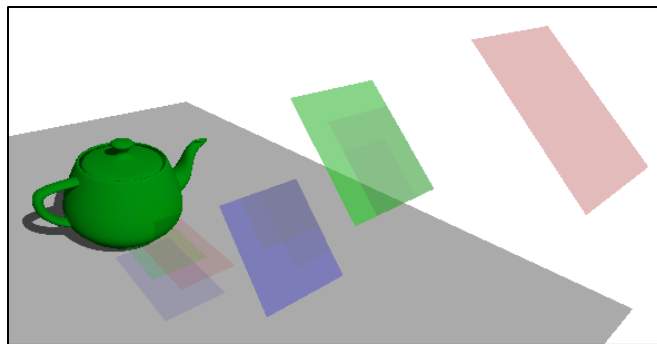
### 2.1.3 Coloured shadow mapping techniques

The following sections detail the two existing approaches being analysed in this report in more detail, so they can be more thoroughly understood, as well as allowing a better comparison with the new technique implemented in this project that they are being evaluated alongside.

### *Translucent shadows*

The first of the two existing techniques that will be explored is translucent shadows as described in Activision Blizzard's 2008 presentation on rendering techniques in StarCraft II, "StarCraft II Effects & Techniques" [2]. The approach sought to allow transparent effects, such as smoke and explosion to cast shadows which only partially occluded light. An example of translucent shadows can be seen in Figure 2.1.3.A.

The effect itself utilises dual shadow maps along with a colour buffer. An initial pass renders a traditional shadow map, rendering a depth map of only the scene's opaque geometry from the perspective of the light. A second render pass renders an additional depth map composed of only transparent geometry, with a less or equal depth test. These depth-only passes effectively record both the closest opaque and closest transparent geometry to the light [2].



**Figure 2.1.3.A.** An example of translucent shadows. Notice the appropriate shadows on the ground plane, teapot, and red square, but also how the green square receives shadows from the blue square despite being closer to the light source than it. It is also not immediately obvious, but the blue square is casting a shadow on itself.

These dual shadow maps are used to subdivide the scene into three regions: geometry fully occluded by an opaque shadow caster and therefore fully in shadow, geometry which is partially occluded by a transparent shadow caster, and unobstructed geometry. The fully occluded region is fully in shadow, as would be the case if only a single shadow map was being used, while the unobstructed geometry is fully lit as would be expected [2].

A third render pass populates a colour buffer (which is cleared to white before rendering) to determine the colour and intensity of the shadows cast on geometry which is partially occluded from the light by transparent surfaces. Transparent geometry in the scene which passes a depth test with the opaque shadow map (i.e., it is between the first opaque surface and

the light) is then rendered closest-to-farthest from the light with each surface filtering the light shining through it by its transmission colour (this could be determined by the colour of the surface itself including its alpha value, as seems to be the case in StarCraft II's implementation, or a separate value as described in McGuire and Enderton's "Coloured stochastic shadow maps" [3]). The final result stored in this buffer is the colour that would result from white light (hence the white clear colour) shining through all transparent geometry that passed the depth test with the opaque shadow map. During the final rendering of the scene geometry, these values are multiplied with the light colour to determine the colour of the lighting illuminating the geometry in the partially occluded region of the scene [2].

This technique is, in theory, very efficient with only two additional render passes required when compared to a traditional shadow map implementation, neither of which uses any particularly expensive operations; however, it does bring with it a few issues.

One complication is the front-to-back ordering required during the third render pass which populates the shadow colour buffer. As previously mentioned, this type of ordering can become expensive in very complex scenes, and it is in addition to the camera-depth ordering that may be required already. A specific method to accomplish this sorting is not described in the text, so will likely vary between specific implementations of this technique.

The choice as to how to accomplish this ordering in this report's implementation is discussed in future sections, but the reason why this ordering must be done, and why it is done front-to-back in this technique merits consideration.

As described earlier in the section on translucency, the order in which translucent surfaces are rendered is vital for the correct rendering of pixels which contain overlapping transparent fragments; however, in the described implementation of translucent shadows, this rendering is done front-to-back as opposed to the usual back-to-front rendering of transparent geometry. This is because the shadow colour pass is essentially doing the reverse of a usual render pass; instead of modelling light shining through the back of translucent surfaces into the camera, it is modelling light shining through transparent surfaces before reaching the other geometry in the scene. The white texture in this case is representative of white light, and by rendering the transparent surfaces front-to-back over the white clear colour, the filtering of white light through those surfaces can be approximated.

An additional complication with translucent shadows as described is the usage of a single colour buffer to determine the lighting colour of the entire partially occluded region. Each texel in the colour buffer can represent an arbitrary number of transparent surfaces, and since this colour is used to determine the lighting contribution to the entire region (including the

transparent surfaces themselves), if two or more transparent surfaces are overlapping from the perspective of the light, the colour of the lighting contribution to some of the transparent surfaces (specifically those occluded by the surface closest to the light) will be incorrectly lit. How conspicuous this issue is depends greatly on the nature of the scene being rendered. For many scenes it is a non-issue, whether because the issue itself does not appear or it is so subtle that it is difficult to notice, but in other scenes the issue is incredibly obvious, with transparent surfaces very distance to the light affecting the shading of surfaces much closer to it.

Lastly, a significant strength of this approach is that, assuming correct rendering order of transparent geometry, this approach will always give you the correct<sup>1</sup> shadow colour on the opaque surfaces receiving them. This is debatably the most important (or at least noticeable) element of translucent shadows in most use cases and is a major boon of this method along with its relatively cheap cost and straightforward implementation.

### *Coloured stochastic shadow maps*

The second technique under consideration is McGuire and Enderton's coloured stochastic shadow maps [3]. The paper introduces two variants of CSSMs: CSSM1 and CSSM2. The CSSM1 algorithm is the generalised form of the technique, allowing for variable types and count of colour channels, while CSSM2 optimises the technique for RGB applications specifically. An example of coloured stochastic shadow mapping can be seen in Figure 2.1.3.B.

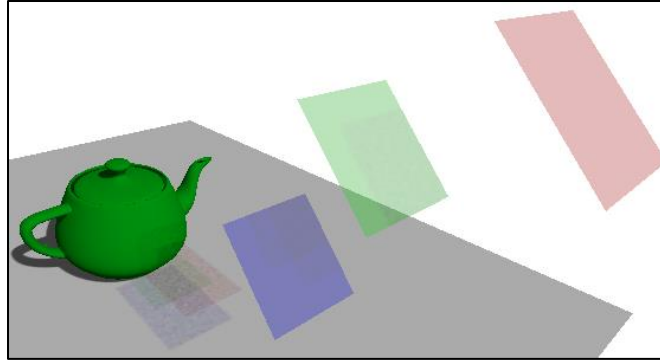
Coloured stochastic shadow maps are an extension of the stochastic shadow maps [11] discussed in the previous section. In fact, the more general algorithm described by McGuire and Enderton, CSSM1, involves rendering an array of stochastic shadow maps for each wavelength of light that is being considered [3]. For most applications, only red, green, and blue light is of concern, which is where the CSSM2 algorithm becomes relevant.

Important to the coloured stochastic shadow mapping technique is the idea of the transmission property of surfaces. A surface's transmission is a collection of values each of which encode the likelihood of the surface to transmit photons of a certain wavelength to be transmitted through the surface. The key difference between the CSSM1 and CSSM2 algorithm is that the CSSM1 algorithm makes no assumption about the number of wavelengths the shadow map is concerned with and therefore accommodates this by utilising a variable number

---

<sup>1</sup> A drawback of translucent shadows is that while the technique does give convincing results, the shadows it produces are not actually physically correct without modifications. This inaccuracy is addressed (though only superficially corrected) in this report's implementation; changes made are described in the Software Implementation chapter.

of (non-coloured) stochastic shadow maps to deal with each specific wavelength. On the other hand, the CSSM2 algorithm leverages the fact that most applications are only concerned with red, green, and blue light to utilise a single colour buffer along with a single depth buffer to encode the information necessary to handle these three wavelengths [3].



**Figure 2.1.3.B.** An example of a CSSM2 shadow map being used to render the teapot scene. Unlike translucent shadows, coloured stochastic shadow maps generate correct shading at each layer of the image, including the transparent surfaces themselves, but introduce noise to the shaded areas.

The CSSM2 algorithm begins by rendering all of the opaque geometry in the scene from the light's perspective, storing its depth value in not only the depth buffer, but each colour channel of the colour buffer. The published approach to generating a CSSM2 shadow map involves rendering the depth information to the depth buffer as with a traditional shadow mapping pass, then rendering a large quad coloured with the depth information to write the depth values to the colour channels [3].

After recording the opaque depth information, the colour blending operator is set to **BLEND\_MIN**, and the blend factors are each set to **ONE** (or the equivalent values for the environments not using OpenGL), allowing an operation similar to a depth test to be performed on information written to the colour channels (since each colour channel is essentially being used like a depth buffer) [3]. With these new settings all of the transparent surfaces are rendered with their output colour being determined by the following equation, where  $\vec{\xi}$  is a vector of three random numbers, and  $\vec{p}$  is the vector containing the likelihood of red, green, and blue light to be transmitted through the surface:

$$colour = \max(depth, (\vec{\xi} > \vec{p})) [3].$$

Additionally,  $\vec{p}$  can be calculated with the following equation where  $\vec{t}$  is the surface's transmission values and  $\alpha$  is the surface's opacity:

$$\vec{p} = \alpha (1 - \vec{t}) [3].$$

These render passes together are modelling a number of things. First the depth pass of the opaque geometry which is copied to all of the colour channels simulates the opaque geometry blocking all wavelengths of light and fully occluding geometry beyond it from the light. The transparent pass is simulating two effects at the same time. The inclusion of  $\alpha$  models the probability of light to hit the surface at all, while  $\vec{t}$  models the chance of light that hits the surface to continue moving through and past it. After these render passes, a coloured stochastic shadow map has been produced with each of its pixel representing the final destination of a single photon cast from the light [3].

CSSMs can be improved using multisampling, as well as other techniques, but the above method is the core of the described implementation.

Just as with non-coloured stochastic shadow maps, coloured stochastic shadow maps must implement a softening technique, such as PCF, to eliminate the noise that stochastic sampling introduces, and just like non-coloured stochastic shadow maps, they require larger PCF kernels than traditional shadow maps; however, since CSSM2 shadow maps are not single-channel depth buffers, but instead three-channel colour buffers, the optimised 2x2 PCF sampling that most hardware provides (such as through GLSL's `sampler2Dshadow`) cannot be used and makes the PCF process more costly for CSSM2 shadow maps in addition to them already needing a larger kernel for acceptable results [3]. CSSM1 shadow maps, being an array of single-channel depth buffers, do not run into this issue, but the CSSM1 algorithm itself is inherently more costly due to using an array of depth-buffers instead of CSSM2's single colour buffer.

Both variants of coloured stochastic shadow maps are attractive due to their being able to render coloured shadows at every layer of the scene, and in CSSM2's case, being able to do so at a cost comparable to a traditional shadow map, but they do introduce a non-negligible amount of noise into the shadows themselves. There are methods to dramatically lessen the noise present in these shadow maps (though not completely eliminate it), and though they tend to either come with more computation cost or introduce bias into the algorithm, they are incredibly effective at reducing the noise present in renders utilising coloured stochastic shadow maps. McGuire and Enderton discuss some of these techniques in the original publication. The renders seen in this report do not implement these denoising technique, so it should be kept in mind the high degree of noise seen in some of the CSSM2 renders in this report is not a given for all implementations utilising it.

## 2.2 Methods and Techniques

In planning how to approach this project, careful considerations were taken to ensure the tools and techniques used were a good fit for the specific requirements of the application as well as in line with industry standards and best practices. A few significant choices made during development are highlighted in this section.

The first major choice to make in this project was what graphics API should be used to support the techniques. Among the papers cited and discussed in this report, OpenGL seemed the popular choice for the contemporary papers, but in the larger context of computer graphics the main options are DirectX, Metal, OpenGL, and Vulkan.

All four of these options still see regular use but do come with their own advantages and disadvantages. DirectX and Metal are platform exclusive, with each only targeting single operating systems (Windows and macOS respectively). OpenGL and Vulkan, both introduced and maintained by the Khronos Group, are each cross-platform, working on most operating systems (though some environments require extra extensions, such as MoltenVK for using Vulkan with macOS), as well as cross-vendor, working on hardware from multiple manufacturers. OpenGL was first released in 1992 and has received numerous updates since, is widely used, and tends to automate and obfuscate many lower-level operations making it simpler to use for some. Vulkan, released in 2016, is a modern API providing more control over lower-level processes on graphics hardware with a lower overhead when compared to OpenGL, taking a direction similar to other newer graphics APIs such as Metal and DirectX12.

The next major choice in this project beyond graphics API was of what renderer to use. The number of renderers, game engines, and other tools that could've been used to facilitate this project's implementation is too numerous to list, but the more manageable choice in this respect is whether it is more beneficial to use an existing renderer or one created specifically for this project. Using an existing option would help with this project's replicability but would most likely introduce overhead beyond what is needed by this project. On the other hand, an original renderer has less guarantees of portability, but can be crafted specifically to suit the needs of this project.

Some other important choices made were over what programming language, what programming environment and related what compiler, as well as what version control software and service were best fits for the project.

## 2.3 Choice of methods

In this report's implementation Vulkan was selected as the API of choice. This choice was made for several reasons, the first is Vulkan's ability to be used across numerous platforms, making this project's results more easily replicable across different environments. Additionally, due to Vulkan being a more modern API, implementing existing techniques with Vulkan, especially when many of the associated publications are focused on OpenGL implementations, seems valuable as it showcases the techniques within a more modern environment.

Moving up a layer from the graphics API is the rendering engine. This project's use of an original renderer was two-fold: using an original renderer meant more familiarity with the environment, speeding up development, and additionally allowed features that were unnecessary to the project to be removed as well as additional ones that were necessary or beneficial to be added, modifications that would've been more difficult if not forbidden when using a publicly available renderer.

All programming done for this project was done in C++, as it is the standard used in most professional game development environments as well as one of the most common used in graphics programming alongside C. Additionally, with C and C++ being very popular languages overall, using C++ meant a large body of third-party software was available to aid in the project's development.

The project itself was developed using Visual Studio 2022 and packaged as a Visual Studio project. This selection was made due to Visual Studio's popularity in hobby, academic, and professional settings, as well as its customisability and myriad of debugging and profiling tools.

As stated previously Git and GitHub were used for version control and to host the source code of the application. They both have seen widespread adoption in many industries and excel with projects which are primarily composed of source files without too many binary assets, so seemed good fits for this application. Perforce Helix Core was also considered for the same purposes, but is more targeted at companies and organisations, and though a free version is available, the overhead of using it over GitHub was prohibitive.

## 3. Software Requirements and System Design

The aims of this report are to gain insight into the performance and results of various coloured shadow mapping techniques, therefore ensuring that the application developed as part of this project is able to support these techniques and evaluate them effectively is of the



utmost importance. As was considered in the earlier section discussing the specific methods and tools chosen for use in this project, it is equally important that the application conforms to best programming practices and is additionally implemented in such a way that it is applicable to modern environments.

### 3.1 Software Requirements

The primary software capabilities required to meet the needs of this project are the ability to render a given scene utilising each of the shadow rendering techniques under consideration. The weight of this requirement falls to the renderer.

The renderer will need to support several features to meet the needs of this project. In particular, it will require the ability to:

- render structured mesh data with support for relevant attributes, materials, and other data (such as textures),
- write rendered output both to intermediate buffers as well as swap chain framebuffers and present data rendered to the swap chain to the screen,
- utilise user-defined shaders with support for defining the behaviour of the vertex and fragment stages at a minimum,
- handle multiple user-defined uniform/descriptor set data structures for use with shaders,
- allow the creation of multiple graphics pipelines with support for user-defined behaviours,
- provide a means for a user to define multiple render passes and execute these passes,
- handle lighting, with support for directional lights at a minimum
- manage the lighting data present in a given scene, with support for sending relevant data to shaders, editing the lighting data in real time, and updating intermediate lighting data in real time when required (such as updating the projection-view matrix of directional lights when the camera moves),
- handle user input, specifically mouse and keyboard input,
- create and manage cameras controllable by user input,
- handle various types of data buffers both on the CPU and GPU, including but not limited to images and mesh data.

Implementing the above features should ensure that the fundamental rendering requirements of this project are fulfilled, but several more peripheral features are required, some of which are implied by the above features but still merit stating:

- the ability to load at least one commonly available 3D model file format (preferably one with support for storing material data) and parse it such that buffers compatible with the renderer can be produced. The GLTF<sup>2</sup> file format is a logical choice, but other formats, such as FBX<sup>3</sup> would also fulfil this requirement,
- the ability to load image files and/or generate image buffers, aside from those loaded as part of model materials, seeing as noise textures will most likely be used in the application, this is a requirement,
- the ability to compile shader programs into the SPIR-V<sup>4</sup> format for use with Vulkan.
- the ability to switch between different render passes or collections of render passes easily; this does not necessarily need to be done at run time, but the ability to easily switch between the different shadow mapping approaches is a must for ease of use, even if implemented through a method such as preprocessor definitions.

These above features together will ensure results are producible, replicable, and easily comparable between different shadow mapping techniques in an efficient manner. These capabilities will allow virtually any scene to be rendered, allowing the shadow quality of each technique as well as their specific capabilities to be analysed and compared. However, these metrics are only part of what is needed to accomplish the goals of this project. The ability to measure the performance of each technique being examined is also crucial. To accomplish this, the application will need the ability to:

- produce timestamps at runtime, GPU timestamps are mandatory, but CPU timestamps are optional, as the performance of techniques on the GPU is the primary focus of this report,
- compare these timestamps in a meaningful way, with measurement strategies tailored to each shadow mapping technique

---

<sup>2</sup> GLTF Overview: <https://www.khronos.org/gltf/>

<sup>3</sup> Autodesk FBX: <https://www.autodesk.com/products/fbx/overview>

<sup>4</sup> SPIR Overview: <https://www.khronos.org/spir/>

- write timing results to a file; CSV files would be a good fit as they are compatible with Microsoft Excel, making the creation of charts much easier and allowing results to be easier communicated.

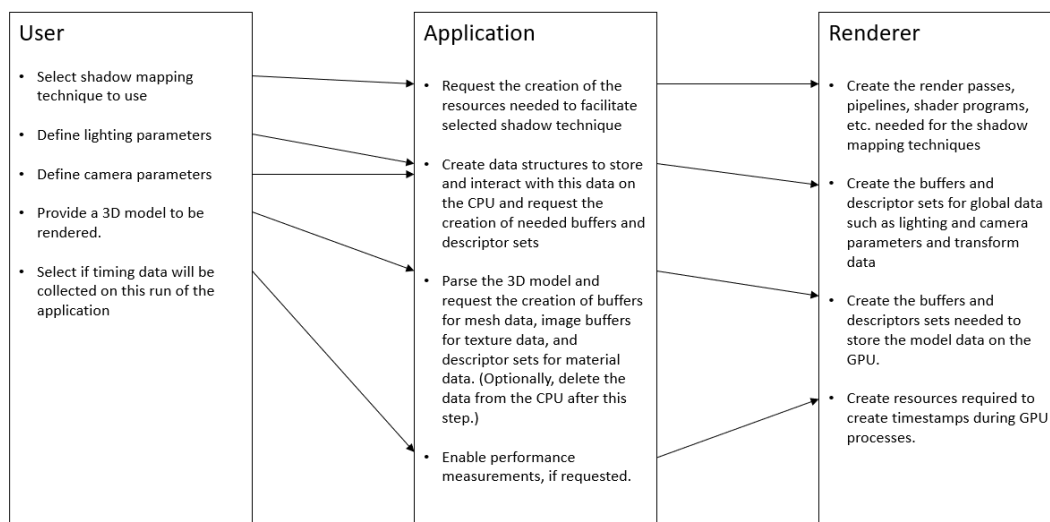
These features will allow the performance measurements needed to fulfil this projects goals to be produced. Additionally, since this project's application implements all techniques under consideration using the same renderer and parameters, and the measurements are all taken on the same hardware in the same conditions, the results will be directly and meaningfully comparable to one another.

### 3.2 System Design

With the above requirements enumerated and justified, a design for how to fulfil them can be produced. The charts shown in figures 3.2.A. and 3.2.B. give a general overview of the responsibilities of different components of the application and how they flow into one another. The enumerated processes are roughly chronological from top-to-bottom within each division, and the arrows represent processes which are either dependent on previous steps being completed or are triggered by them.

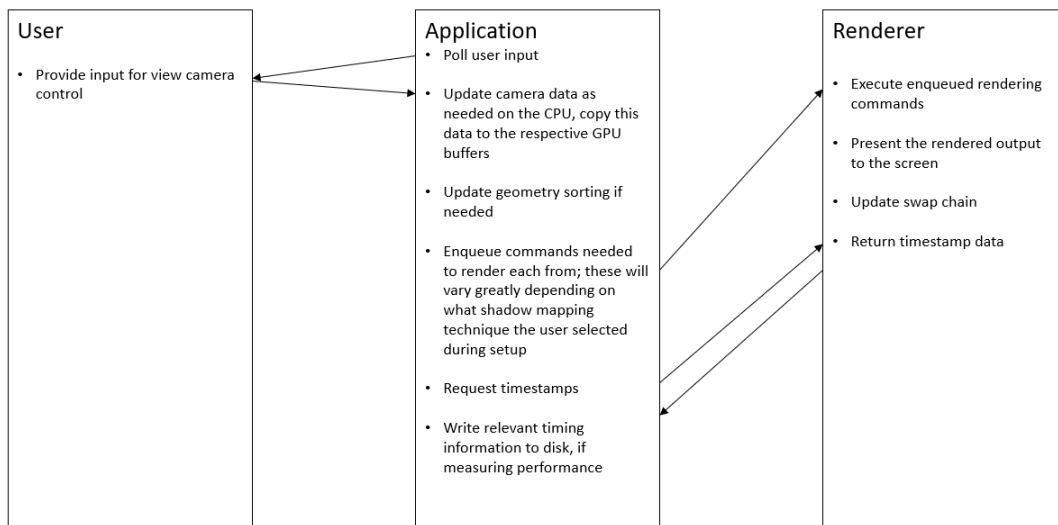
These diagrams are very high-level and general, only meant to communicate the relationship between the different elements of the application. The implementation of specific shadow mapping techniques and evaluation methods are described in more detail in the following chapters.

#### Setup



**Figure 3.2.A.** A diagram showing a high-level overview of the distribution of responsibilities and general flow of the application during its setup phase.

### Runtime / Application Loop



**Figure 3.2.B.** A diagram showing a high-level overview of the distribution of responsibilities and general flow of the application within its main loop.

## 4. Software Implementation

### 4.2 Shadow mapping technique implementations

As the detailed processes of major existing techniques have already been described in detail, the following sections on translucent shadows and coloured stochastic shadow maps instead focus on the modifications and adaptations made in this report's implementations of them, as well as the specific resources and rendering requirements of the implementations in a Vulkan environment.

The final section focuses on the new approach introduced in this report, composited translucent shadows, discussing its development, design, and implementation.

#### 4.2.1 Translucent shadows

##### *Modifications*

Notably absent in the published description of translucent shadows was the technique used to order the transparent surfaces being rendered to the colour buffer in the technique's third render pass. This is an understandable exclusion; in most cases the sorting is a simple task

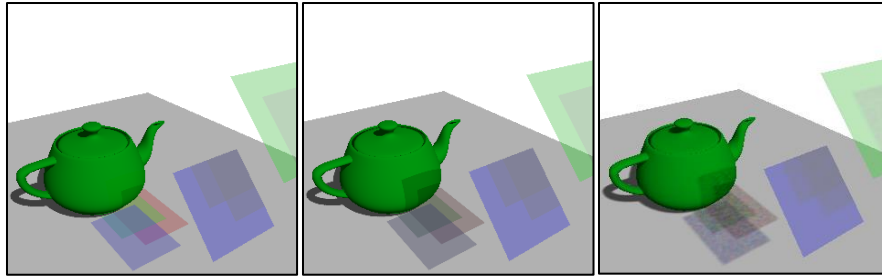
and the best method to accomplish it depends on the application implementing it. There are several techniques that could be used to accomplish this ordering, but it is achieved in this report's implementation by sorting the geometry at the mesh level based upon the centre of each mesh's bounding box. This technique does come with issues, particularly when dealing with objects of varying sizes or objects which both occlude each other in different regions of their geometry, but this technique is among the cheapest that could be used and still guarantee passable ordering, which is why it was selected. The application does this sorting on the CPU, and it is only done once at startup since the light is assumed static in this project's implementation.

It is worth noting that this same type of sorting is also used to sort the transparent geometry itself for rendering from the camera's perspective; this is true not only of the implementation for this technique, but of all of the implementations in this project. This choice was made in large part to make the results more directly comparable to each other.

Another notable aspect of this technique's implementation is a modification which was made to its shadow colour pass. When rendering transparent geometry to the shadow colour buffer, the published method describes simply rendering the surfaces front-to-back from the light's perspective to the shadow colour map. This does give passable results, but without modifications it is not possible to accurately model transmission in this manner, as it will be treated as equivalent to surface colour. This can lead to renders that look unrealistic; for example, this is what causes the lack of shadow cast on the green pot by the green plane in the left-most render of figure 4.2.1.A. To improve the results, I took inspiration from the CSSM2 light transmission probability equation by considering surface transmission as a separate property from surface colour and incorporated the alpha value into the colour contribution of the transparent surfaces during the coloured shadow pass. The equation used is as follows, with  $\alpha$  representing the surface's opacity and  $\vec{t}$  representing the surface's light transmission:

$$\text{shadow colour} = (1 - \alpha) * \vec{t}.$$

The second images in figures 4.2.1.A and 4.2.1.B. demonstrate the results of this modification, compared to both unmodified translucent shadows (the left and top images) and CSSM2 shadows (the right and bottom images). In these renders, the CSSM2 shadows and modified translucent shadows treat the surface transmission as half of the surface colour.



**Figure 4.2.1.A.** A demonstration of the coloured shadows cast on coloured and non-coloured surfaces by (from left to right) translucent shadows, modified translucent shadows, and CSSM2.



**Figure 4.2.1.B.** A demonstration of the intensity of shadows cast by (from top to bottom) translucent shadows, modified translucent shadows, and CSSM2.

### Resource requirements

The logical resources and buffers required by this technique are as follows:

- **VkRenderPass** suitable for rendering a Williams shadow map
- **VkRenderPass** suitable for rendering the shadow colour map
- Two depth buffers of the desired shadow map size (**VkImage** and **VkImageView**)
- One colour buffer of the desired shadow map size (**VkImage** and **VkImageView**)
- **VkFramebuffer** suitable for rendering a Williams shadow map, using the depth buffer as an attachment
- **VkFramebuffer** suitable for rendering the shadow colour map, using both the depth buffer and colour buffer as attachments

- A **VkSampler** suitable for sampling a shadow map (i.e., **VkSamplerCreateInfo** with **compareEnabled = VK\_TRUE**)
- A **VkSampler** suitable for sampling a colour buffer
- **VkDescriptorSet** suitable for sending three image samplers (opaque depth map, transparent depth map, shadow colour buffer) and a uniform containing the shadow view-projection matrix.
- Two **VkPipeline** with associated **VkPipelineLayout**: one for rendering the depth maps and one for rendering the shadow colour buffer

The rendering requirements of this technique are as follows:

- A render pass to render the opaque geometry depth buffer (opaque only)
- A render pass to render the transparent geometry depth buffer (transparent only)
- A render pass to render the shadow colour buffer (transparent only)

#### 4.2.2 Coloured stochastic shadow maps

##### *Modifications*

This project's implementation of CSSM2 is very close to the published approach. The only major change is in the step where the depth information of the scene's opaque geometry is copied from the depth buffer to each colour channel of the colour buffer. In the described approach this is done by rendering a quad textured by the depth buffer, which seems to imply an additional render pass being used due to the rendering setting changes necessary to accomplish the copying in this manner (disabling depth write, enabling colour write). In this project's implementation writing to the depth buffer and colour buffer is done in the same pass, removing the need of copying the values over.

Additionally, in this report's application, the implementations of both coloured and depth-only stochastic shadow maps utilise a noise texture to acquire the pseudorandom numbers required to facilitate these approaches.

##### *Resource requirements*

The logical resources and buffers required by this technique are as follows:

- **VkRenderPass** suitable for rendering the opaque depth information and the coloured stochastic shadow map itself
- One depth buffer of the desired shadow map size (**VkImage** and **VkImageView**)
- One colour buffer of the desired shadow map size (**VkImage** and **VkImageView**)
- **VkDescriptorSet** suitable for sending two image samplers (opaque depth map, coloured stochastic shadow map) and a uniform containing the shadow view-projection matrix.
- Two **VkPipeline** with associated **VkPipelineLayout**: one for rendering the opaque depth information and one for rendering the coloured stochastic shadow map itself.
- An additional **VkImage** and **VkImageView** may be needed depending on how the pseudorandom numbers are acquired for the stochastic sampling. This report's implementation used a noise texture (and hence needed these additional resources).

The rendering requirements of this technique are as follows:

- A render pass for rendering the opaque depth information to the depth and colour buffers as well as rendering the coloured stochastic shadow map (these two steps can and should be performed in the same render pass)

#### 4.2.3 Composited translucent shadows

The final coloured shadow mapping technique under consideration is composited translucent shadows, the novel technique introduced by this report. The approach is heavily built upon translucent shadows [2], and with this technique the opaque geometry in the scene is rendered exactly as it is with Filion, et al.'s approach, but multiple render passes of the transparent geometry in the scene are performed, and their results composited over the render of the opaque geometry. This allows composited translucent shadows to overcome the most significant drawback of translucent shadows, their inability to correctly shade transparent surfaces when multiple overlap from the perspective of a light. An example of composited translucent shadows can be seen in Figure 5.2.3.A.

A major strength of translucent shadows is their ability to render convincing, coloured shadows on opaque surfaces, appropriately affected by each coloured surface occluding them from the light. This strength can be leveraged to correctly shade transparent geometry too by partitioning the scene's transparent geometry into different regions and rendering each region individually with its own shadow map. In this way layers of the scene are each treated within

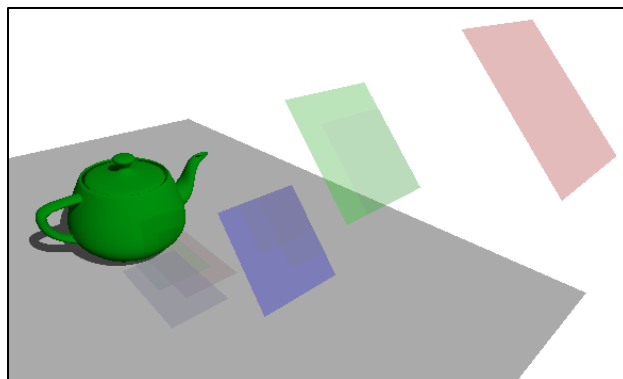


their own render passes, similarly to how translucent shadows treat the scene's opaque geometry.

The major issue that comes with this technique is how to partition the scene. The partitioning method has two requirements:

1. no geometry which occludes other geometry from the light may be in the same partition as the geometry it occludes,
2. the partitions must be separated in such a way that they can be composited over each other and the prior render of the opaque geometry with correct alpha blending.

These requirements must hold for a given partitioning method in order for it to correctly render the scene. The first requirement must hold true, because translucent shadows are still being used to shade each individual partition of the scene, so if a region contains geometry which occludes other geometry in the same partition from the light source, it cannot be correctly shaded, just as with original translucent shadows. The second must hold, because even if each partition itself is rendered correctly, if the individual renders cannot be combined with each other and/or the render of the scene's opaque geometry, a complete render cannot be reconstructed.



**Figure 5.2.3.A.** The teapot scene rendered with composited translucent shadows. Notice the similarity to Fillion, et al.'s translucent shadows, but in this render, the green translucent surface does not receive shadow contributions from the blue surface as it would with the original translucent shadows.

These requirements are non-trivial, and the implementation used in this project's application scales poorly for even mildly complex scenes (and additionally is not completely robust) but does prove that the technique is able to correctly shade all opaque and transparent geometry in a scene with appropriately coloured shadows and without the artifacts present in alternative techniques. Other techniques were tested to fulfil the requirements; depth peeling

was given particular attention, and its application for this purpose is discussed in the next section, but a partitioning technique which was efficient and generally applicable was not found.

The partitioning algorithm used to facilitate composited translucent shadows is of particular importance, because each partition produced by the algorithm will itself incur most of the cost of rendering a scene with translucent shadows. Filion et al.'s translucent shadows are incredibly efficient, as can be seen in the results of this report, however, if a partitioning algorithm performs poorly, the amount of computation required to correctly render a scene can quickly become impractical for real-time applications.

However it is performed, once this partitioning and separate rendering of each region are completed, the renders themselves are composited from back-to-front overtop the render of the scene's opaque geometry using regular alpha blending (i.e., the blending method described in the section on transparency), producing a render with correct translucent shadows on not just opaque geometry but all layers of translucent geometry as well.

#### *Leveraging sorted geometry for partitioning*

The original implementation of translucent shadows requires the ability to render the transparent geometry in a scene ordered by its distance from the light. Additionally, the chosen method for rendering correct transparency in this report's application was to sort transparent geometry based on distance from the camera. Together, these sorted sets were successfully used as a partitioning method for this report's implementation of composited translucent shadows.

Each transparent mesh in the scene requires two render passes. Since meshes are sorted by their proximity to the light, it is simple to render only the meshes between the mesh currently being rendered and the light source, for the purpose of populating the second depth (i.e., first-transparent-surface depth) and shadow colour buffers. This approach allows additional writes to the first-opaque-surface depth buffer to be avoided, as simply leaving it populated with the values from rendering the opaque geometry in the scene suffices. Additionally, because this approach renders surfaces in order of distance from the camera, the final render of each shaded mesh can be drawn directly on top of the previous passes (including the initial opaque geometry pass). This means that no intermediate colour buffers need to be stored, greatly reducing the memory requirements for complex scenes. The only special consideration needed to accomplish this is to ensure the colour buffer being used to store the final results is only cleared once before the initial rendering of the scene's opaque geometry.

While this approach works, it's scaling is far from optimal, as can be seen in the results in the next chapter; however, as stated before, it does work as a proof of concept to show that composited translucent shadows are a viable technique.

### *Depth peeling as a partitioning method*

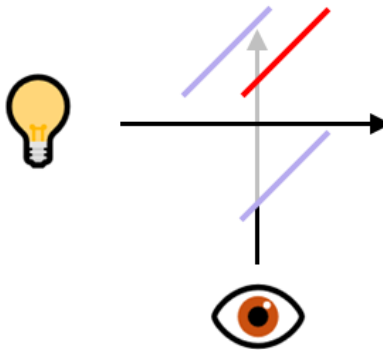
The method that was given the most consideration for use as a partitioning method was depth peeling [12]. In this approach, each layer extracted through depth peeling would be treated as a single partition. In its own render pass, the layer would be treated as the first-opaque-surface in the translucent shadows algorithm and would be the only geometry rendered in the pass rendering scene geometry. This means all transparent surfaces between it and the light would cast shadows on it, but only it would appear in the output to be composited. This report's exploration failed to produce a working partitioning method using depth peeling; depth peeling's use for this type of space partitioning is non-trivial, if even possible, but it can at first glance seem a good fit, so its shortcomings are discussed.

To fulfil either of the set-out partitioning requirements individually with depth peeling is trivial. As depth peeling extracts geometry which overlaps from a given perspective into separate render passes, the first requirement can be fulfilled by performing depth peeling from the perspective of the light, and the second can be fulfilled by depth peeling from the camera's perspective. Both of these techniques were considered, but both come with cases that cause them to fail to fulfil both partitioning requirements.

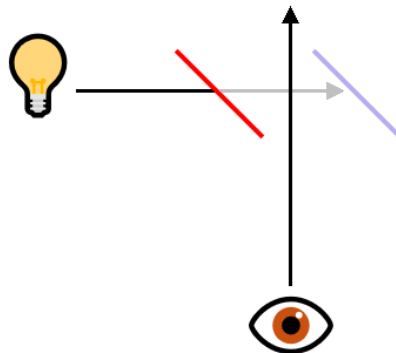
Figure 5.2.3.B. shows a case where depth peeling from the perspective of the light would produce two rendered layers that would be impossible to composite. The blue surfaces would both be extracted in the same layer when depth peeling, and the red surface would be extracted in the next. This poses a problem, because from the camera's perspective portions of the red surface are between portions of the two blue surfaces, but since the two blue surfaces would be rendered to the same image, it would be impossible to perform the compositing necessary to achieve correct transparency. This case would violate the second partitioning requirement, making this approach unfeasible.

Figure 5.2.3.C. demonstrates an example where camera-space depth peeling would make it impossible to correctly shade all surfaces extracted in a layer. Two surfaces which do not overlap from the camera's perspective, can occlude each other from the light's perspective. In the case presented in the figure, the red surface fully occludes the blue surface from the perspective of the light, while these surfaces would be extracted to the same layer when depth peeling. Since the layers extracted when partitioning are treated as opaque geometry for the

purposes of rendering the associated shadow map, in cases such as this, the surfaces closest to the light would shade the occluded surface as if it was fully opaque. Additionally, even in an application that only treated the back-most surface of the layer as opaque geometry, you would run into the usual issues of incorrect shading of intermediate transparent surfaces that regular translucent shadows run into. Cases like this show that camera-space depth peeling violates the first partitioning requirement.



**Figure 5.2.3.B.** An example of a case where two layers peeled from the light's perspective (represented by the light bulb) would be impossible to composite correctly. The blue surfaces would be depth peeled in one pass, and the red surface would be extracted in the next; however, this is an issue as, from the camera's perspective (represented by the eye), the red surface needs to be rendered between the two blue surfaces.



**Figure 5.2.3.C.** An example of a case where camera-space depth peeling would fail. The red and blue surfaces are completely separate from the camera's perspective, and thus would be extracted in the same depth peeling pass, but they are completely overlapping from the light's perspective. Without additional steps undertaken the blue surface would be shaded as completely shadowed by the red surface, as both would be treated as opaque geometry for the pass shading them.

Additionally, even if the problematic cases mentioned are ignored, depth peeling still has other issues as a partitioning method. Both approaches mentioned above require transformations of geometry from light-space to camera-space and vice versa. When rendering

the layers extracted from light-space depth peeling, the geometry that actually needs to be rendered from the camera's perspective would need to be deduced from only a light-space depth buffer, a process which is likely to introduce error. Similarly, when depth peeling from the camera's perspective, the first-opaque-surface depth buffer used to render the shadow map, would have to be deduced from the camera-space depth buffer, which would run into similar issues.

Another issue with light-space depth peeling specifically is that in the very common case that a light is outside the camera's view frustum, the layers produced have a high likelihood themselves to be outside the camera's view frustum, as surfaces closer to the light, not the camera, are rendered first. This means this approach will potentially produce numerous layers that do not contribute to the scene. Worse, this means stopping conditions usually used with depth peeling, such as using a fixed count of layers, could potentially cause some or even all of the transparent geometry in the scene to not be rendered. This is also an issue when depth peeling is used to render transparency from the camera's perspective, but when depth peeling is done from the camera's perspective, each layer rendered reduces the impact of subsequent layers due to the alpha blending equation mentioned in the section on transparency. The diminishing returns of deeper layers means it's less of an issue if they are excluded from a render, but light-space depth peeling does not have this benefit.

Overall, this exploration suggests that depth peeling, while great for its intended purpose of achieving order independent transparency, is likely to be unfeasible as a general-purpose partitioning method for a composited translucent shadows implementation, at least when applied in the manners described in this section.

## **5. Software Testing and Evaluation**

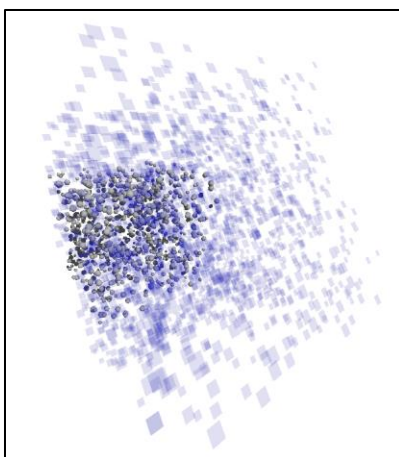
In this chapter, the performance of several of the discussed shadow mapping techniques is measured and compared. The techniques under consideration include the three coloured shadow mapping techniques that have been discussed in this report: translucent shadows [2], CSSM2 [3], and composited translucent shadows. Additionally included are two non-coloured shadow mapping techniques: original Williams' shadow mapping [7] and stochastic shadow mapping [11]; these two techniques were included in order to provide insight into the overhead of coloured shadow mapping techniques over depth-only shadow mapping techniques.

## 5.1 Evaluation Method

The primary performance metric used in this report is the time taken to render a scene with varying amounts of transparent geometry, which is both casting shadows (except in Williams' shadow mapping) and receiving shadows.

Figure 5.2.A demonstrates the scene being used to take the measurements that follow. Every render during this evaluation included 1000 randomly placed opaque, grey spheres, these both serve the purpose of receiving shadows, as well as simulating a realistic scene which is populated with various opaque geometry. Another region of the scene overlapping with the region of opaque spheres, but also extending further towards the light source, contains 2000 translucent, blue quads. The number of blue quads which are enabled for rendering varies depending on the sample, allowing an analysis of each shadow mapping technique's performance in relation to scene complexity.

The lighting used in the scene is a single white directional light that's bounds are updated every frame that the camera moves, so that it approximately covers the camera's view frustum.



**Figure 5.2.A.** An example of the scene used to measure each of the techniques' performance scaling with varying amounts of scene geometry.

The time measurements used in this report are made through Vulkan queries<sup>5</sup> setup to record timestamps<sup>6</sup> [16]. Four different primary measurements were taken on a per-frame basis:

---

<sup>5</sup> `VkQueryPool` reference page: <https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VkQueryPool.html>

<sup>6</sup> `VkCmdWriteTimestamp` reference page: <https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/vkCmdWriteTimestamp.html>

total frame time, time taken to render shadow map(s), time taken to render the scene itself, and combined render time. To measure time taken to render the shadow map(s) required for each technique, a timestamp was requested before the first Vulkan command dealing with shadow map rendering, and another was produced after the last. The same approach was taken for rendering the scene geometry. The total frame time was measured in a similar way, except the time stamps were requested before the very first and after the very last commands of any type enqueued each frame (with the exception of the resetting of the `VkQueryPool` managing the Vulkan queries themselves). For all of the techniques except composited translucent shadows, the combined render time is simply the sum of shadow map rendering time and scene rendering time.

The exception to these measurement approaches is the composited translucent shadows implementation. In the case of composited translucent shadows, the commands responsible for rendering shadow maps and scene geometry are not grouped together; the scene's opaque geometry and the shadow maps used to shade it are rendered in the same way all geometry is rendered in the usual translucent shadows approach, but the number of render passes used to render transparent geometry varies, and this make it difficult to capture this technique's shadow map and scene render times in a fixed amount of timestamps. Instead, this technique measures only shadow map and scene render time of the opaque geometry for those metrics, but instead of simply summing these values for the combined render time, timestamps are taken before and after the iterative rendering of transparent geometry, and this sum is added with the earlier measurements from the rendering of the opaque geometry to calculate the final combined render time.

Because of the discrepancies in measurement of shadow map and scene render time metrics between different techniques, they are not focused on when comparing shadow mapping techniques to each other. The frame time is measured in exactly the same manner across all techniques, but it includes the time taken to execute processes unrelated to the shadow mapping techniques such as rendering the final colour buffer to the swap chain, several image layout transitions, and the updating of uniform buffers. Instead, the combined render time is used as the primary metric of performance, as it gives the best picture of each technique's actual rendering performance in a way that is comparable across techniques. It should be noted that all timings taken for this report only include time spent executing relevant commands on the GPU, so time spent on CPU operations is not reflected in the results displayed in this report.

Aside from timed performance, visual quality of shadows as well as the handling of multiple layers of transparency are considered. These latter two aspects are difficult to

numerically measure, so they are instead tested by comparing renders of scenes meant to highlight them. Details of these implementations are discussed along with the results.

## 5.2 Testing Hardware

To allow for better contextualisation of the results presented in this report, all measurements were taken on a desktop using an AMD Ryzen 7 5800X processor along with an NVIDIA GeForce RTX 3070 GPU.

The GPU used in this setup supports Vulkan timestamps with nanosecond precision, so though they are displayed in milliseconds, the timing results presented in this report were measured at a much higher resolution. The full-precision results are available in the directory containing the application's source code.

## 5.3 Results

Shadow mapping techniques were measured on three metrics, time spent rendering scenes of varying complexity, shadow quality, and handling of layered transparency. The first metric is discussed for all five techniques under consideration, while the latter two are only discussed for techniques that produce coloured shadows.

### 5.3.1 Performance

The combined render times of each approach when rendering the testing scene with varying counts of transparent meshes follow. Figure 5.3.A. shows timings from Williams' shadow mapping [7], figure 5.3.B. from Filion, et al.'s translucent shadows [2], figure 5.3.C. from Enderton et al's stochastic shadow maps [11], and figure 5.3.D. from McGuire and Enderton's CSSM2 [3]. Figures 5.3.E. and 5.3.F. show timings from composited translucent shadows in varying degrees of granularity. The lower resolution of results and inclusion of two graphs of varying granularities are a result of the poorer scaling of the partitioning method used when compared to the scaling of the other shadow mapping approaches; this poorer performance meant a closer analysis of render times when rendering less transparent geometry provided a more meaningful insight.

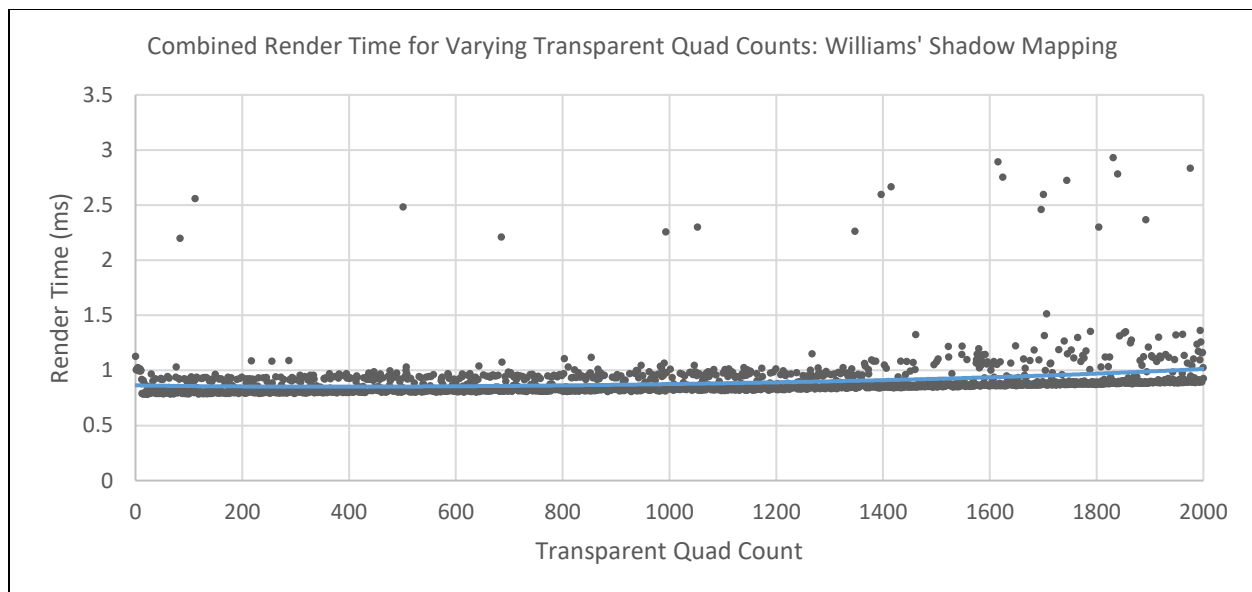
The analysis of performance is broken down into three categories based upon algorithms that scale similarly. Translucent shadows and Williams' shadow maps are discussed first, and have the best scaling with scene complexity, both approaches utilising stochastic



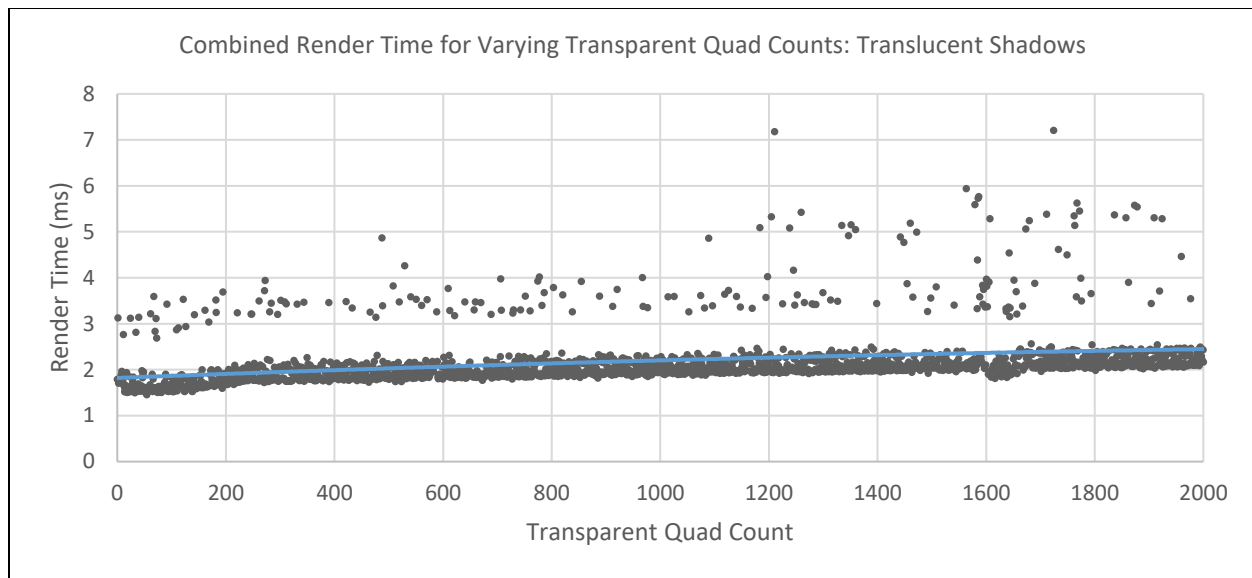
sampling scale worse but not significantly so, and the final section covers translucent shadows, scaling the worst with scene complexity.

Williams' shadow mapping is unsurprisingly the most efficient of the algorithms. The overhead to add Williams' shadows to an environment without shadows is a depth-only render of the scene to a separate single-channel image buffer, which is then sampled and depth tested against once for each fragment rendered. More samples and depth comparisons can be done to implement shadow softening techniques such as PCF. The implementation used when taking the measurements shown in figure 5.3.A. utilised 2x2 hardware PCF through `sampler2Dshadow`.

The performance results for translucent shadows can be seen in figure 5.3.B. While taking longer to render than Williams' shadows, translucent shadows scale similarly and are the second cheapest algorithm tested, taking approximately one millisecond longer than Williams' shadows at all amounts of geometry tested. On one hand, this is surprising, as translucent shadows require more render passes than the stochastic approaches that were also tested, but the render passes themselves are much simpler. In Williams' approach, the scene is divided into two regions, that which is occluded from the light and that that is not. Translucent shadows accomplish subdivision of the scene into three regions by performing a similar operation for both transparent and opaque geometry. Translucent shadows also render an additional pass of a subset of the scene's transparent geometry to a colour image buffer. In this way, translucent shadows may require more operations than Williams' shadow mapping, but the operations themselves are similar, leading to the similar scaling with complexity and similarly cheap cost overall.



**Figure 5.3.A.** Timings recorded when using Williams' shadow mapping with a 2x2 hardware PCF kernel.



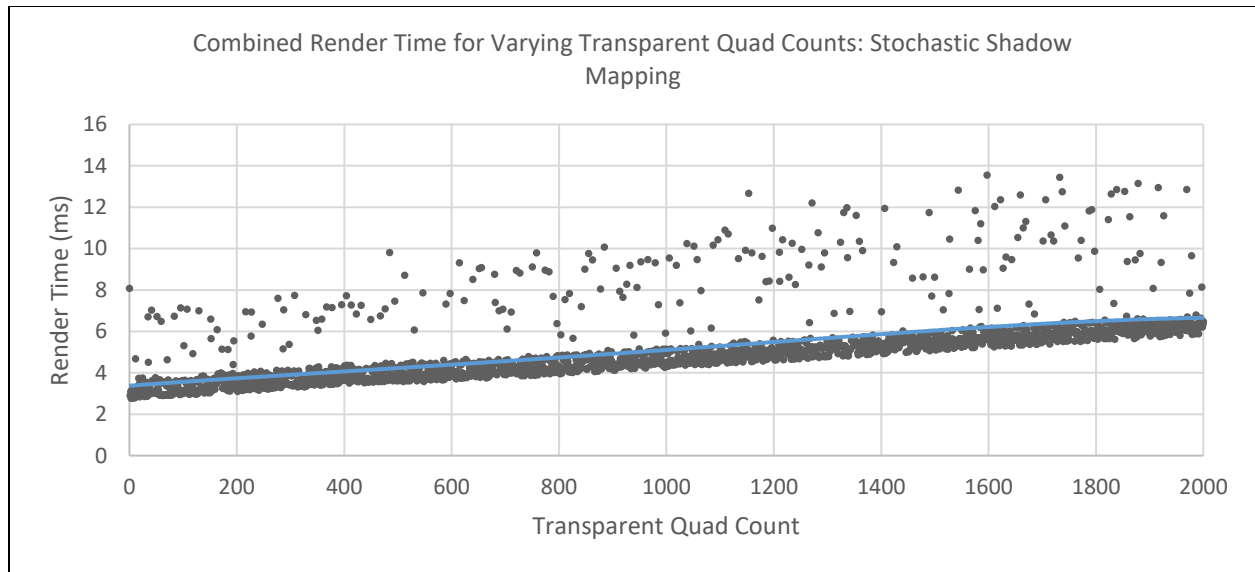
**Figure 5.3.B.** Timings recorded when using translucent shadows with a 2x2 hardware PCF kernel.

The stochastic approaches each have a higher baseline cost and scale worse with scene complexity than Williams' shadow mapping and transparent shadows. Despite the fewer number of render passes and less resources required by the stochastic approaches when compared to translucent shadows, the major differences that incur the additional computation cost are: a need for random number generation, the need for a much larger PCF kernel, and the stochastic discarding of fragments when rendering the shadow map.

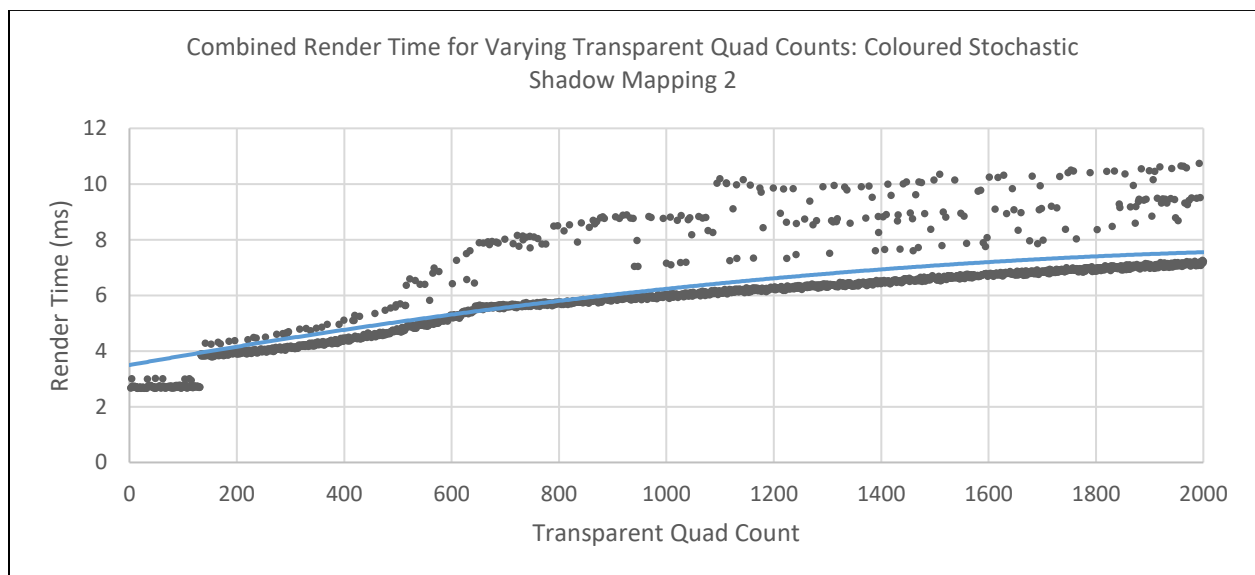
In an attempt to keep runtime costs low, a noise texture was sampled from to emulate random numbers; while sampling from a noise texture is certainly much quicker than generating pseudo random numbers in a shader, it still requires an extra texture sample for each fragment when rendering the shadow maps (stochastic shadow maps and CSSM2 require one and three random numbers respectively, but an RGB noise texture was used during evaluation, allowing three pseudorandom numbers to be acquired per sample). Because the option of generating pseudo random numbers in the shader was avoided in this implementation, the cost of generating random numbers is low, but this may not be true of all applications employing stochastic sampling.

The stochastic discarding of fragments when rendering depth-only stochastic shadow maps could also be a contributing factor to its worse scaling. Discarded fragments that are being processed in parallel could have their threads forced to wait until nearby undiscarded fragments are done rendering; however, while this is an issue that arises in some settings, I do

not think this is a large contributor to overall performance in this technique's case considering the overall cheap cost of the shader performing the stochastic discarding. Additionally, since this discarding is only done when rendering non-coloured stochastic shadow maps, and they scale similarly to the CSSM2 approach, this additionally suggest that this is not a major contributor to computation time.



**Figure 5.3.C.** Timings recorded when using stochastic shadow mapping with 2x2 hardware PCF used in addition to a 9x9 kernel implemented in the fragment shader.



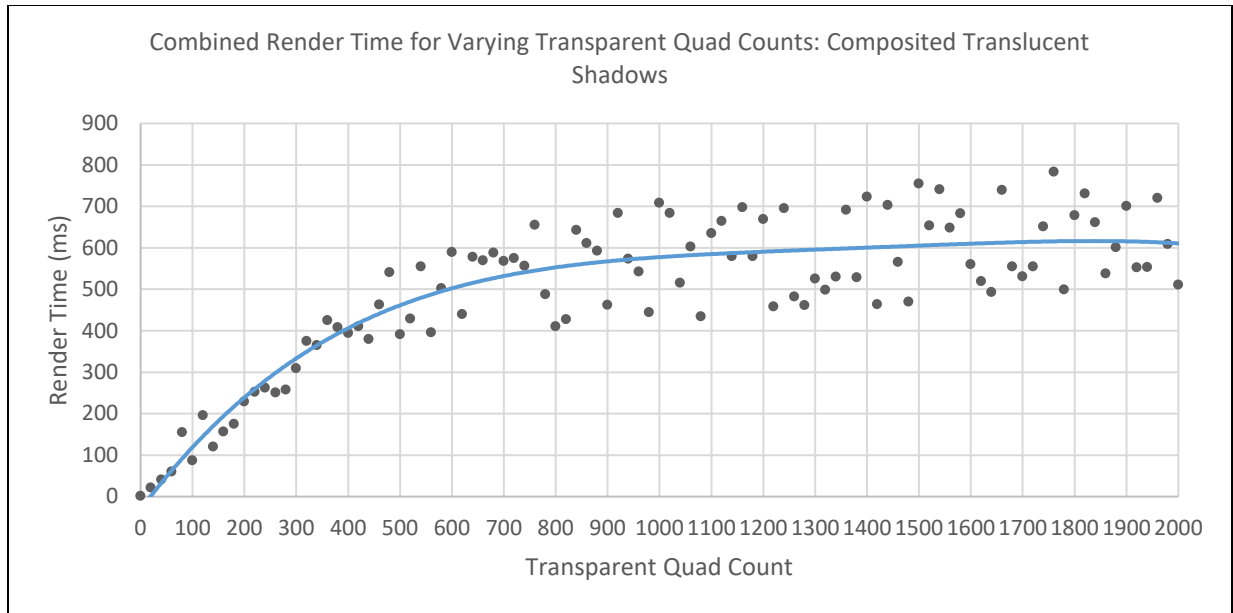
**Figure 5.3.D.** Timings recorded when using stochastic shadow mapping with a 9x9 PCF kernel implemented in the fragment shader.

Another consideration is the larger PCF kernel required by the stochastic approaches, which in my testing was a significant cost. An additional set of timing measurements was taken for the CSSM2 approach where the 9x9 kernel used for PCF was reduced to a 3x3 kernel; this data is available along with the rest of the results. With lower amounts of transparent geometry in the scene, the measurements with the 3x3 kernel were nearly the same as with the 9x9 kernel, but the scaling with growing amounts of geometry was significantly better when using the 3x3 kernel. When nearing on 2000 transparent surfaces, the run using the 9x9 kernel was well over seven milliseconds per frame, but the 3x3 kernel instance only just went above five milliseconds. This suggests that the larger PCF kernel is a large part of why the stochastic shadow mapping approaches experience a more steeply increasing cost with scene complexity when compared with Williams' shadow maps and translucent shadows.

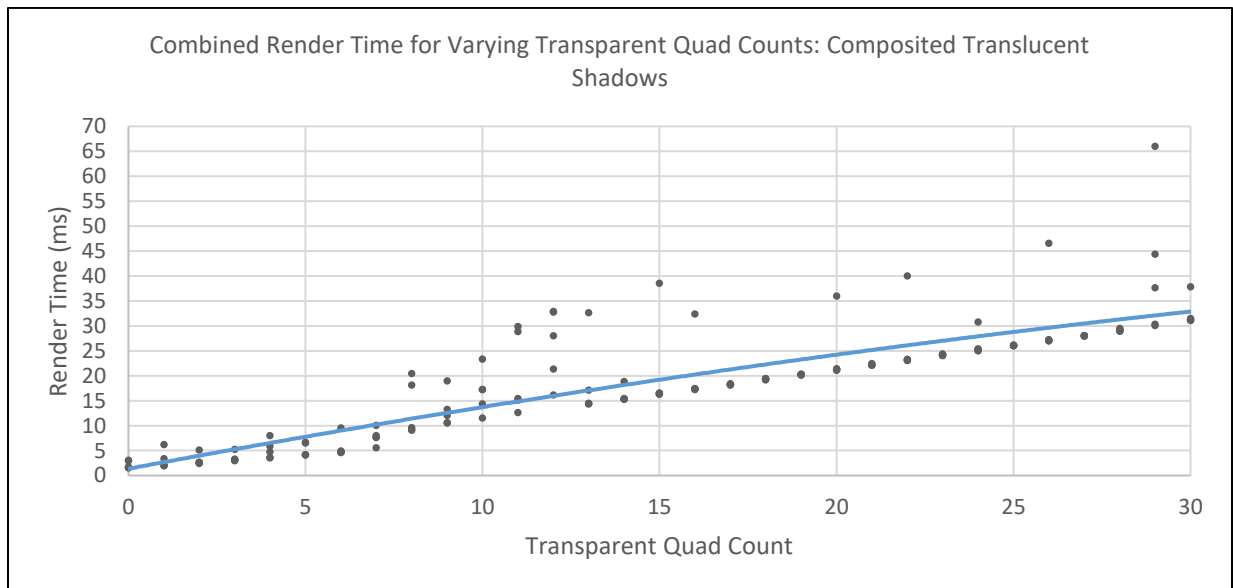
On the topic of percentage-closer filtering, it is also worth mentioning that all techniques discussed are able to utilise hardware PCF, except CSSM2, which is much cheaper than manually implementing the filtering in a shader, though the larger kernels needed by the stochastic techniques means stochastic shadow mapping will more than likely need manual PCF on top of the hardware PCF, though a smaller kernel can be used as a result of the inclusion of hardware PCF.

Last among the techniques being considered is composited translucent shadows, which had by far the sharpest increase in cost with scene complexity. As the primary factor contributing to the cost of this technique is the count of regions that scene geometry is divided into, with each region requiring a comparable amount of computation as rendering a whole scene with translucent shadows, the poor scaling of this technique is largely a result of the partitioning method used. Since every transparent mesh was considered a division of its own, this meant two additional render passes were needed for each transparent object added to the scene, resulting in the sharp increase in cost seen in the data. With better partitioning of the scene these results could, in theory, be drastically improved.

Even with a far suboptimal partitioning approach, the technique is still capable of running at speeds suitable for interactive environments when only being used to render small amounts of transparent geometry, and therefore if this technique is reserved only for a few, important transparent elements in a scene, and a cheaper technique is used to render the rest, this technique still shows promise, particularly due to the quality of its renders, which is discussed in the following sections.



**Figure 5.3.E.** Timings recorded when using composited translucent shadows with a 2x2 hardware PCF kernel, for between 0 and 2000 transparent meshes, with samples taken at 20 mesh increments.



**Figure 5.3.F.** Timings recorded when using composited translucent shadows with a 2x2 hardware PCF kernel, for between 0 and 30 transparent meshes, with five samples at every mesh increment.

### 5.3.2 Shadow Quality

While it is hard to quantify the visual appearance of shadows, it still merits discussion, especially as results are debatably the most important aspect of the techniques. Important to note is that in this section translucent shadows and composited translucent shadows will be

lumped together, as in ideal conditions they will produce identical shadows; this is discussed in more detail in the next section discussing how each technique handles translucent geometry that overlaps from the light's perspective.

The image chosen to test shadow quality is of the Strasbourg rose window. The image contains areas of small- and large-scale details in a multitude of colours and shades, making it a good overall choice for benchmarking the quality of the techniques. The rose window image can be seen in figure 5.3.2.A., while renders produced by shining a white light through a quad textured with the image can be seen in figure 5.3.2.B.

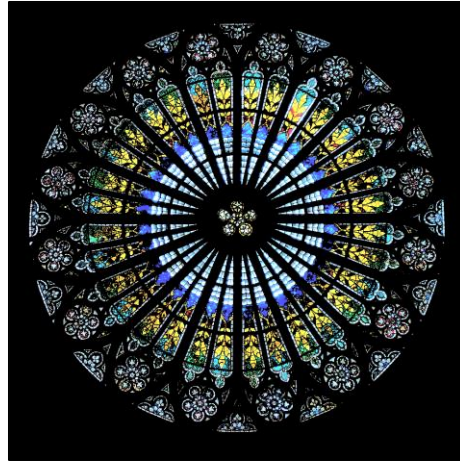
The clear victor for image clarity are the translucent shadow techniques. They produce no noise or other artifacts in their renders, which allows a very small PCF kernel to be used if one is used at all. While PCF is great for softening shadows and decent for de-emphasising noise, it does cause fine details to be lost, so the ability to produce satisfactory results with a smaller kernel size is incredibly valuable. Additionally, regions of the same colour are uniform in the renders with translucent shadows, and the divisions between different elements in the image are clear.

On the other hand, the stochastic noise present in the CSSM2 renders is quite visible, even in the render using a 9x9 PCF kernel. The noise is especially noticable in regions which are bright but have low saturation, which is to be expected given how the technique simulates light transmission. The boundaries between different regions of the image are also blurred, but most of this information is still retained and visible, even if not as sharp as in the translucent shadow renders.

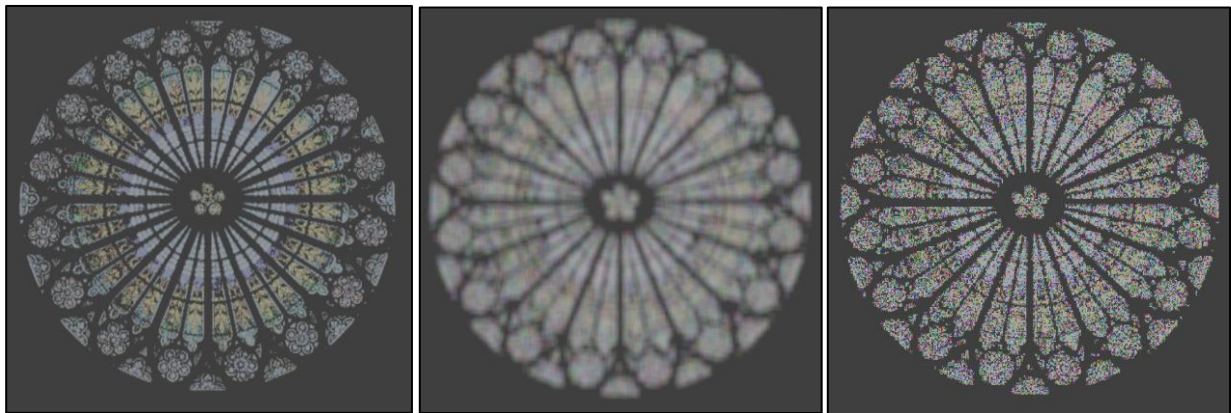
All of this said, this example of coloured stochastic shadow mapping is a bit of a worst case scenario. There are techniques that can be utilised to reduce the noise produced by stochastic shadow mapping techniques with less loss in visual clarity than simply using a larger PCF kernel, which were not employed in the renders presented here; however, without the ability to render infinitely many samples there will still be some degree of noise present. Many of these noise-reduction techniques are discussed in the original publication for coloured stochastic shadow maps [3].

Another consideration that is not as immediately obvious, is that coloured stochastic shadow maps produce much more realistic results and their results produce a better approximation of light transmission as the amount of samples taken, both during the rendering of the shadow map and when sampling from it, is increased. While the translucent shadow techniques may look more clear and do produce convincing results, they do not approach

results that are as physically correct, so if this is of major concern in a particular application, they are not well suited for it.



**Figure 5.3.2.A.** The image of the Strasbourg rose window that was used to test shadow quality. The black regions are fully opaque while the coloured regions are translucent.



**Figure 5.3.2.B.** Shadows cast by shining white light through the image in Figure 5.3.2.A. onto a white surface. From left-to-right, translucent shadows/composited translucent shadows (both techniques will have identical results in this case), CSSM2 with a 9x9 PCF kernel, and CSSM2 without PCF. The slightly faded look of the renders is a combination of the scene's light not shining directly on the surface and the ambient light of the scene washing out the darker colours.

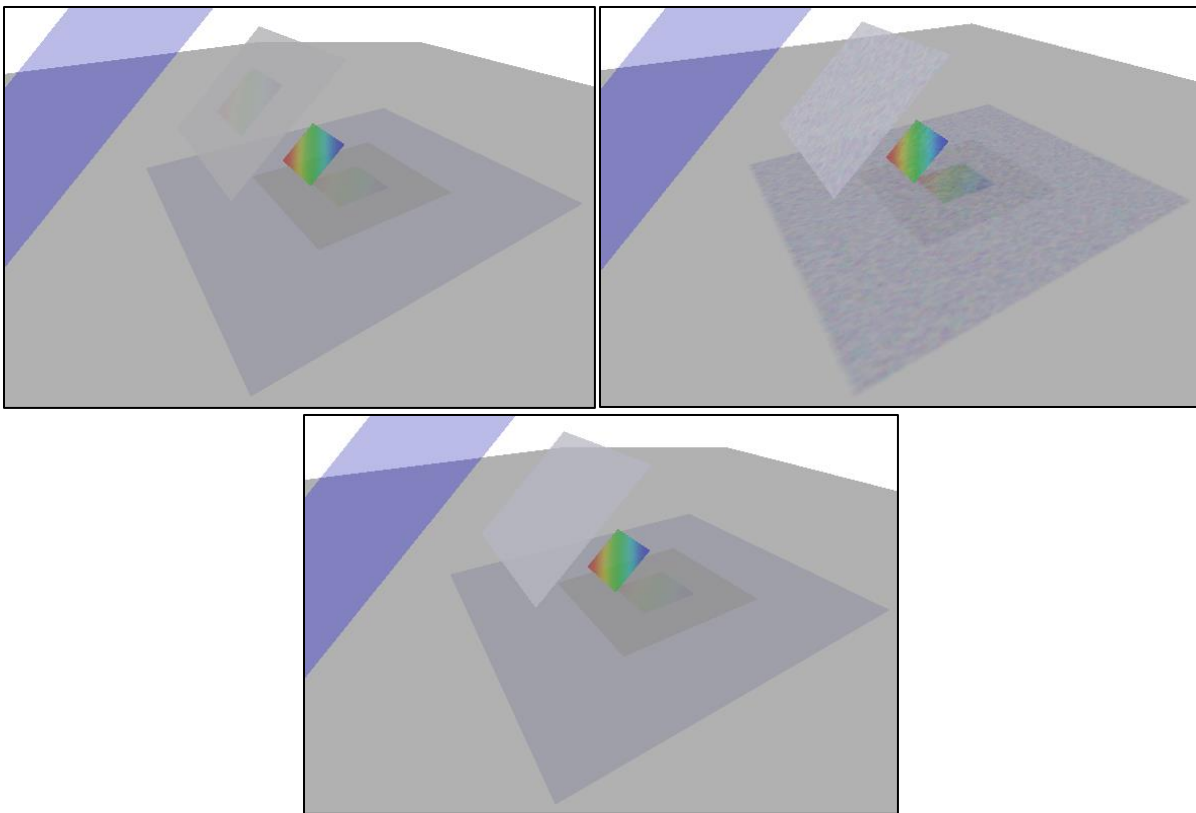
### 5.3.3 Layered Transparency

The final metric being considered is how each technique handles shading transparent geometry which overlaps from the light's perspective, that is, how well it can produce multiple layers of transparency.

To test this, a scene was set up where a white directional light source was shone through blue, grey, and rainbow transparent quads onto an opaque white quad. The results can be seen in figure 5.3.3.A.

While translucent shadows correctly shade both the closest transparent and opaque surface to the light source, it does not shade any transparent surfaces beside the first correctly. This manifests as the rainbow quad casting a shadow onto the grey quad, which is closer to the light source than it, as well as the darker appearance of the grey quad, which is a result of the grey quad casting a shadow onto itself. With translucent shadows, all surfaces occluded by the closest transparent layer to the light, but in front of the closest opaque layer from the light will receive the same shadow contributions.

On the other hand, CSSM2 and composited translucent shadows correctly shade all transparent surfaces, just in the same manner as they do the opaque surface that ultimately stops the light.



**Figure 5.3.3.A.** Three renders of a scene showing the resulting shadows cast on both transparent and opaque surfaces when a white light is shown through the transparent surfaces, showing renders produced with translucent shadows (top-left), CSSM2 (top-right), and composited translucent shadows (bottom).



## 6. Conclusions and Future Work

### 6.1 Conclusions

Based on the observations made, it is clear that each of the three coloured shadow mapping approaches excel in different areas, meaning which technique is best in a given setting depends on the specific requirements it comes with.

If performance is the primary concern, Filion, et al.'s translucent shadows [2] are by far the best choice, with both the quickest baseline performance of the techniques analysed as well as the best scaling with scene complexity. McGuire and Enderton's CSSM2 [3] does have a higher baseline cost and slightly worse scaling than translucent shadows, but still performs well enough to comfortably fit into most real-time settings. Performing by far the worst in this metric is composited translucent shadows, which were only able to handle single digit counts of transparent meshes at real-time framerates when utilising the described partitioning algorithm.

If visual quality is the primary concern, composited translucent shadows or translucent shadows are the best choice, depending on the specific requirements of the application. Composited translucent shadows, while the most expensive, also offer the best visuals overall, producing appealing results without artifacts, and also allowing for many correctly shaded transparent layers. Translucent shadows and coloured stochastic shadow maps each lack in one of these areas, with translucent shadows lacking the ability to support multiple layers of transparency, and coloured stochastic shadow maps containing noise from the stochastic sampling the approach is built upon (though this noise can be dramatically lessened if additional denoising techniques are applied). If multiple layers of transparency are not needed, translucent shadows are the best choice, as when there are no transparent surfaces that overlap from the light's point of view, translucent shadows can produce identical shadows to composited translucent shadows in significantly less time,

If physically correct shadows are needed, coloured stochastic shadow maps provide the best results, as they allow surface transmission to be modelled in the most realistic way of all the techniques discussed.

The results gathered have confirmed that despite their lack of widespread adoption, CSSM2 and translucent shadows are both algorithms capable of running at real-time framerates, even in complex scenes, while providing appealing results.

Composited translucent shadows are similarly able to produce attractive results, though are only suitable for use on simple scenes or subsets of more complex scenes, performing poorly when used as the sole method of shadow rendering in a complex scene.

## 6.2 Future Work

The partitioning method used to facilitate composited translucent shadows was simply ordering transparent meshes based on their proximity to the camera and the shadow-casting light source, and while this worked as a proof of concept it is incredibly inefficient. Depth peeling was explored as a possible approach to divide the scene more efficiently for the purposes of rendering composited translucent shadows, but ultimately proved difficult to adapt for the purpose, if it is possible to utilise it to this end at all. This leaves a vacancy for a more efficient general partitioning algorithm. Additionally, since the number of layers the scene is divided into when rendering composited translucent shadows is the primary determining factor of the technique's complexity, an improved algorithm could allow composited translucent shadows to be applicable in more general rendering settings.

Additionally, a partitioning algorithm for composited translucent shadows does not necessarily need to be general. An environment where transparent meshes are arranged in a way that they can be exploited for an easier partitioning process, such as if many transparent meshes are in a uniform layout, could allow for a more specialised algorithm to be used, making composited translucent shadows a more viable approach. Identifying and developing partitioning algorithms for these cases is another area that merits some exploration.

While this report solely examined shadow mapping techniques, a similar performance comparison between these techniques and non-shadow mapping techniques capable of achieving similar results, such as ray tracing, would be a worthwhile topic to consider.

## References

- [1] W. Reeves, D. Salesin and R. Cook, "Rendering Antialiased Shadows with Depth Maps," *ACM SIGGRAPH*, vol. 21, no. 4, pp. 283 - 291, 1987.
- [2] D. Fillion, R. McNaughton and N. Tatarchuk (Editor), "Starcraft II Effects & Techniques," *Advances in Real-Time Rendering in 3D Graphics and Games Course - SIGGRAPH 2008*, pp. 133-164, 2008.

- [3] M. McGuire and E. Enderton, "Colored stochastic shadow maps," *Symposium on Interactive 3D Graphics and Games*, pp. 89-96, 2011.
- [4] Epic Games, "Unreal Engine 5.0 Documentation," [Online]. Available: <https://docs.unrealengine.com/5.0/en-US/>. [Accessed 11 July 2023].
- [5] Unity Technologies, "Unity Manual," 2023. [Online]. Available: <https://docs.unity3d.com/Manual/index.html>. [Accessed 11 July 2023].
- [6] F. Crow, "Shadow Algorithms for Computer Graphics," *Computer Graphics (SIGGRAPH Proceedings)*, vol. 11, no. 2, pp. 242 - 248, 1977.
- [7] L. Williams, "Casting Curved Shadows on Curved Surfaces," *ACM SIGGRAPH Computer Graphics*, vol. 12, no. 3, pp. p. 270-274, 1978.
- [8] D. Shreiner, G. Sellers, J. Kessenich and B. Licea-Kane, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*, Ann Arbor, Michigan: Pearson Education, Inc., 2013.
- [9] J. Boksansky, M. Wimmer, J. Bittner, E. Haines (editor) and T. Akenine-Möller (editor), "Ray Traced Shadows: Maintaining Real-Time Frame Rates," in *Ray Tracing Gems*, Berkley, California, Apress, 2019, pp. 159 - 182.
- [10] A. R. Smith, "Alpha and the History of Digital Compositing," [alvyray.com](http://alvyray.com), 1995.
- [11] E. Enderton, E. Sintorn, P. Shirley and D. Luebke, "Stochastic Transparency," *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 2D Graphics and Games*, pp. 157-164, 2010.
- [12] C. Everitt, "Interactive Order-Independent Transparency," *NVIDIA*, white paper, 2001.
- [13] D. Wexler, L. Gritz, E. Enderton and J. Rice, "GPU-Accelerated High Quality Hidden Surface Removal," *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pp. 7 - 14, 2005.
- [14] L. Bavoil and K. Myers, "Order Independent Transparency with Dual Depth Peeling," *NVIDIA*, technical report, 2008.
- [15] H. Fuchs, J. Goldfeather, J. P. Hultquist, S. Spach, J. D. Austin, F. P. Brooks, J. G. Eyles and J. Poulton, "Fast spheres, shadows, textures, transparencies, and image enhancements in pixel-planes," *ACM SIGGRAPH Computer Graphics*, vol. 19, no. 2, p. 119, 1985.

[16] "Vulkan API Reference Pages," Khronos Group, 4 August 2023. [Online]. Available: <https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/>. [Accessed 11 August 2023].

## Appendix A: Third-Party Software

All third-party software utilised in the project application, along with licenses when appropriate, can be found within the directory titled `ext` within the project source code.

- **GLFW:** <https://www.glfw.org/>
  - Used to facilitate window creation and handling, along with input polling.
- **GLM:** <https://github.com/g-truc/glm>
  - Used for much of the vector and matrix arithmetic in the project.
- **labutils:** not publicly available, modified and used with permission from the author, Dr Markus Billeter.
  - Used in a modified form to help easier utilise the Vulkan graphics API.
- **shaderc:** <https://github.com/google/shaderc>
  - The glslc tool that is included in this project is used to compile GLSL shaders into SPIR-V shaders.
- **stb:** <https://github.com/nothings/stb>
  - Used to facilitate image file loading and parsing.
- **TinyGLTF:** <https://github.com/syoyo/tinygltf>
  - Used to facilitate the loading and parsing of GLTF 2.0 files.
- **Volk:** <https://github.com/zeux/volk>
  - Meta-loader used to ease the use of the Vulkan graphics API.

- **Vulkan:** <https://www.vulkan.org/>
  - Modern, cross-platform graphics API used as the foundation for the graphics in this project.
- **Vulkan Memory Allocator:** <https://gpuopen.com/vulkan-memory-allocator/>
  - Buffer and image memory allocation library for use with Vulkan.

## Appendix B: Ethical Issues

GitHub was used for version control of this project's source code to ensure a central and protected copy of the project was available at all times and to ease development of the project.

The application associated with this project is comprised both of original code, as well as third-party code, libraries, and tools. In Appendix A, third-party resources used are listed along with their original sources. Additionally, when applicable, licenses have been included along with the associated files of third-party resources within the application's GitHub repository.

There are some resources that are exceptions. Notably, the files in the *labutils* directory are not publicly available and therefore do not have a license, but are used with permission from the author, Dr Markus Billeter, in modified form. Additionally, the rendering engine used in this project's application is an original one, built on a more rudimentary renderer I developed during my enrolment in the High-Performance Graphics module at the University of Leeds. This partial reuse was cleared with the module leader, and seemed appropriate as this report's focus is not on the renderer, but instead on the shadow mapping techniques themselves, with the renderer being a means to this end.

## Appendix C: Project GitHub Link and Raw Evaluation Data

The source code of this project as well as the raw performance data measured can be found at this web address: <https://github.com/citrus-tree/MScProject>.